Engenharia de recursos

Transforme dados brutos em recursos valiosos com Feature Engine

Casa do Código | alura

THIAGO BENEVIDES

Sumário

- ISBN
- Agradecimentos
- Sobre o Autor
- Sobre o livro
- Introdução à engenharia de recursos
- PARTE 1: Trabalhando com valores ausentes
  - 1 Trabalhando com valores ausentes em um conjunto de dados
  - 2 Projeto prático — Substituição de valores ausentes
- PARTE 2: Trabalhando com valores numéricos
  - 3 Trabalhando com valores numéricos
  - 4 Projeto prático — Discretização de valores numéricos
- PARTE 3: Padronização de recursos
  - 5 Padronização de recursos
  - 6 Projeto prático — Padronização de valores de ações da Bolsa
- PARTE 4: Codificando valores categóricos
  - 7 Codificação de valores categóricos
  - 8 Projeto prático — Codificação de variáveis categóricas
- PARTE 5: Redução de dimensionalidade, upsampling, downsampling e métricas para avaliação
  - 9 Trabalhando com redução de dimensionalidade, upsampling e downsampling
  - 10 Projeto prático — Redução de dimensionalidade com upsampling e downsampling
- Apêndice
  - 11 Apêndice: Qual é o melhor modelo?

ISBN

Impresso: 978-85-5519-393-4

Digital: 978-85-5519-394-1

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Benevides, Thiago
Engenharia de recursos : transforme dados brutos em recursos valiosos com Feature Engine / Thiago Benevides. -- 1. ed. -- São Paulo : AOVS Sistemas de Informática, 2025.

Bibliografia.
ISBN 978-85-5519-393-4

1. Ciência da Computação 2. Dados - Análise 3. Dados - Estruturas (Ciência da computação) 4. Estruturas de dados (Computação) 5. Linguagem de programação (Computadores) I. Título.

24-245407 CDD-005.73

Índices para catálogo sistemático:

1. Dados : Estruturas : Processamento de dados 005.73

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Estou extremamente feliz e agradecido por ter concluído este livro e gostaria de expressar minha gratidão a todas as pessoas que me apoiaram nessa jornada. Em primeiro lugar, agradeço a Deus e aos meus pais, Maria e João, que sempre me encorajaram e me apoiaram em meus sonhos e projetos.

Em segundo lugar, gostaria de dedicar este livro à minha madrinha de batismo, Regina, que me incentivou a sempre buscar a excelência e o aprendizado, pois eles são a única forma de superação e de se alcançar algum futuro viável. Sou imensamente grato por sua confiança em mim e pela sua influência positiva em minha vida.

Também quero agradecer à minha orientadora de TCC, a professora Nara Boko, que me ensinou a importância da pesquisa e como manter o foco em projetos de longo prazo. Foram dois anos de trabalho árduo, correções e aprendizado, mas, graças a ela, aprendi habilidades valiosas que utilizei na elaboração deste livro.

Por último, mas não menos importante, gostaria de agradecer à Vivian Matsui pela sua dedicação e compreensão na edição deste livro. Ela me guiou com paciência e profissionalismo para que o meu sonho se tornasse real. Se vocês estão conseguindo ler este livro hoje, parte desse mérito é dela, que conseguiu extrair o meu melhor durante o processo de escrita. Vivian, obrigado!

Sobre o Autor

Thiago Benevides é licenciado em Matemática pela Universidade Tecnológica Federal do Paraná e atualmente cursa pós-graduação em Estatística Aplicada e um MBA em Gestão Financeira e Análise de Risco, ambos pela Faculdade Anhanguera. Ele possui diversos cursos de programação e análise de dados na Udemy, além de um curso de visão computacional disponível na Hotmart.

Atualmente, atua como professor de matemática no ensino médio e também ministra cursos voltados à educação financeira e à otimização de portfólios, com enfoque em estatística, programação e machine learning.

Sobre o livro

Decidi escrever este livro porque percebi uma escassez de material em português sobre esse tema que, para mim, é crucial para um bom desenvolvimento de modelos de *machine learning*, isto é, modelos eficientes e livres de enviesamento. Fui inspirado a escrever por um livro chamado *Mulheres Invisíveis: O viés dos dados em um mundo projetado por homens*, da escritora Caroline Criado Perez. A leitura desse livro me mostrou como um mau gerenciamento de recursos para uma posterior análise pode reforçar preconceitos e até mesmo causar a segregação. Acredito que a chave para evitar esses problemas é ser crítico em relação aos dados usados para treinar nossos modelos de *machine learning*.

Neste livro, utilizo a linguagem de programação Python 3, e os códigos foram escritos no Google Colaboratory. Entretanto, você pode utilizar o Anaconda ou outro editor de texto de sua preferência. Os códigos com que trabalharemos podem ser encontrados no repositório do GitHub: https://github.com/ThiagoBenevides/feature-engine-transformando-dados-brutos-em-recursos.

Para quem é o livro

Este livro foi elaborado para pessoas que já possuem conhecimentos básicos em Python e análise de dados e desejam aprofundar seus conhecimentos em engenharia de recursos. Com a engenharia de recursos, é possível aprimorar ainda mais seus modelos de *machine learning*, resultando em uma melhor compreensão dos seus dados e maior precisão dos seus modelos.

Porém, caso você ainda não possua conhecimento em Python e análise de dados, não se preocupe! Para que você consiga acompanhar melhor o conteúdo deste livro, recomendo dois livros também da Casa do Código que vão lhe ajudar a se familiarizar com esses temas. O primeiro é o *Python: Escrevendo seus primeiros programas*, do Felipe Cruz, que vai lhe apresentar os conceitos

básicos da linguagem Python e ajudar você a escrever seus primeiros códigos. O segundo é o livro *Machine learning: Introdução à classificação*, de Guilherme Silveira e Bennett Bullock, que aborda os conceitos fundamentais de *machine learning* e como aplicá-los. Com esses dois livros em seu arsenal, você estará bem-preparado(a) para se aprofundar ainda mais em análise de dados e aproveitar ao máximo o conteúdo deste livro.

BIBLIOTECAS UTILIZADAS

Como as bibliotecas em Python estão em constantes mudanças, vamos descrever aqui as bibliotecas que vão ser utilizadas ao longo deste livro para que você não tenha conflito com versões:

- pandas versão: 2.14
- yahoo finance: 0.2.44

As bibliotecas citadas no decorrer deste livro foram atualizadas na versão mais recente do Google Colaboratory, entretanto, no repositório deste livro, há um arquivo de nome `requirements.txt` que facilita a instalação das versões exatas utilizadas nos exemplos. Para garantir a compatibilidade, basta fazer o upload desse arquivo para o *Google Colaboratory* e executar o seguinte comando:

```
!pip install requirements.txt
```

Com isso, as versões instaladas das bibliotecas ficarão alinhadas às utilizadas no livro, assegurando a execução correta dos códigos.

Introdução à engenharia de recursos

O termo **aprendizado de máquina**, ou ***machine learning***, está cada vez mais popular. Startups e empresas utilizam esse recurso para conseguir extrair *insights* ou fazer previsões a respeito de algo, por meio de algoritmos baseados em modelos matemáticos e estatísticos.

Por exemplo, suponha que trabalhamos em uma loja que serve cafés e acompanhamentos. Poderíamos usar as informações dos nossos clientes para entendermos o padrão de compra e quais tipos de cafés mais saem e em quais dias e, com esses dados, preparar nossos estoques e até mesmo oferecer descontos ou novos tipos de acompanhamentos de acordo com o gosto de nossos clientes.

Eu utilizei o café, e não algo tecnológico, porque eu quero que logo neste começo você já consiga compreender que *machine learning* e *análise de dados* não estão somente no mundo da tecnologia. Onde quer que existam dados e sua coleta, é possível desenvolver a análise de dados e até mesmo um algoritmo de *machine learning* para prever situações futuras.

Porém, nem tudo são flores: se por algum motivo você utilizar dados ruins para treinar seus modelos ou até mesmo para fazer suas análises, você poderá chegar a conclusões equivocadas, o que pode levar você a ter prejuízo, ao invés de ter lucro.

Então, antes de utilizar esses dados para o que quer que seja, você precisa saber trabalhar neles, seja ajustando-os, seja criando novos dados a partir dos dados existentes. Isso tem um nome: chama-se ***feature engine*** ou, em português, **engenharia de recursos**.

A engenharia de recursos é uma das etapas fundamentais no desenvolvimento de projetos de *machine learning*, pois é através dela que conseguimos transformar e extrair informações relevantes dos dados brutos, de forma a criarmos novas variáveis, que também são chamadas de recursos.

Em *machine learning*, um recurso (também conhecido como *feature*) é uma medida ou característica que descreve um determinado objeto ou evento de interesse. Esses recursos são utilizados para alimentar os modelos de aprendizado de máquina, que aprendem a partir de características e, desta forma, conseguem realizar suas previsões. E esses recursos podem ser de várias formas, por exemplo: imagens, textos ou valores numéricos.

A principal vantagem de utilizarmos a técnica de *feature engine* é a melhoria do desempenho dos modelos de *machine learning*. Ao criarmos recursos mais significativos e relevantes, é possível aumentarmos a precisão e a acurácia dos algoritmos de aprendizado de máquina. Além disso, a engenharia de recursos pode ajudar a lidar com problemas de dimensionalidade, levando a um modelo mais eficiente e robusto.

Outra vantagem já citada, mas que vale ressaltar, é a de que a engenharia de recursos pode nos ajudar a evitar falsas análises e equívocos ao extrairmos informações relevantes e significativas dos dados. Por exemplo, ao analisarmos os dados de transições financeiras a fim de se evitar fraudes, podemos criar novos recursos que capturem características específicas das transações suspeitas. Esses novos recursos podem ajudar a identificar padrões e características únicas que indiquem uma transação fraudulenta, evitando falsos positivos e aumentando a eficácia do modelo.

Como o livro está estruturado

O livro foi dividido em 5 partes, cada uma dedicada a um tópico relevante para engenharia de recursos. Em cada uma das partes, você vai encontrar uma explicação sobre o assunto para compreender melhor esse tópico e, em seguida, você terá um projeto prático, onde aplicaremos as bibliotecas *scikit-learn* e *feature-engine* em um conjunto de dados maior.

Parte 1: Trabalhando com valores ausentes

Nesta parte, vamos discutir os processos de substituição de valores ausentes em um conjunto de dados. Veremos como substituir valores ausentes em um conjunto de dados numéricos utilizando os métodos de substituição pela média e depois pela mediana. Em seguida, abordaremos os valores ausentes em um conjunto de dados categóricos, aprendendo a substituir os valores ausentes pela moda.

Por fim, vamos explorar os processos de substituição de valores ausentes em um conjunto de dados sobre o Titanic. Veremos como utilizar a biblioteca *scikit-learn* e a *feature-engine*.

Parte 2: Trabalhando com valores numéricos

Nesta parte, aprenderemos sobre a técnica de discretização de valores numéricos, que consiste em transformar um conjunto de dados contínuos em um conjunto com menos valores distintos. Discutiremos os diferentes métodos para a discretização, por exemplo, por faixa de valor igual e frequência; além disso, veremos dois métodos não supervisionados, que são os *k-means* e a *árvore de classificação*

Para finalizar essa parte, utilizaremos os dados do Titanic para aplicar as técnicas de discretização que vimos, mais especificamente, veremos o processo de discretização na idade dos passageiros.

Parte 3: Padronização de recursos

Nesta parte, abordaremos a importância da padronização de recursos em um conjunto de dados, bem como as diferentes técnicas de padronização disponíveis, como a normalização, média e min-max.

Para finalizar essa parte, vamos aplicar a padronização em um conjunto de dados que coletaremos da Bolsa de Valores de São Paulo, a Bovespa, com auxílio da biblioteca *Yahoo Finance*. Além

disso, construiremos um modelo preditivo utilizando o modelo ARIMA para avaliar a qualidade das transformações realizadas e verificar se a padronização foi efetiva na melhoria do modelo.

Parte 4: Codificando valores categóricos

Nesta parte, vamos explorar técnicas de codificação de valores categóricos, que consiste em transformar esses valores em valores numéricos para que sejam compatíveis com algoritmos de *machine learning*.

Para finalizar essa parte, vamos aplicar as técnicas de codificação. Vamos usar o conjunto de dados do Titanic para fazermos a aplicação da codificação, mais especificamente a codificação na variável *gênero*; para isso, utilizaremos as bibliotecas *scikit-learn* e a *feature-engine*.

Parte 5: Redução de dimensionalidade, *upsampling*, *downsampling* e métricas para avaliação

Nesta parte, aprenderemos técnicas para avaliar a qualidade de nossos dados e reduzir a quantidade para aumentarmos a velocidade de treino do nosso modelo e, para isso, usaremos a técnica conhecida como PCA. Além disso, veremos como lidar com o problema de dados desbalanceados por meio das técnicas de *upsampling* e *downsampling*.

Apêndice

Por fim, veremos a importância da avaliação da precisão dos modelos de *machine learning*, além de técnicas para mensurar a qualidade dos modelos de regressão e de classificação.

PARTE 1: Trabalhando com valores ausentes

CAPÍTULO 1
Trabalhando com valores ausentes em um conjunto de dados

> O notebook dos dados deste capítulo e do projeto prático podem ser baixados em: materiais da parte 1 (https://github.com/ThiagoBenevides/feature-engine-transformando-dados-brutos-em-recursos/tree/main/parte_um)

Quando falamos em dados ou coleta de dados, temos que ter em mente que muitos dos processos hoje são feitos de forma automatizada, e essas automações estão suscetíveis a falhas durante o processo. Essas falhas podem provocar a perda de dados ou até mesmo a não coleta, fazendo com que tenhamos em nosso conjunto de dados o que chamamos de **valores ausentes**.

Um valor ausente exibe um valor que foi ignorado ou que não foi coletado de forma correta, o que pode gerar erros em nossas análises ou ocasionar problemas como *overfitting* e *underfitting* na hora de ajustarmos o nosso modelo de *machine learning*. Para que você consiga entender melhor o que um valor ausente pode provocar, veja a seguinte tabela:

Índice	Idade	Gênero	Tipo de dor	Pressão sanguínea	Colesterol
1	40	M	??	140	289
2	49	??	NAP	??	180
3	37	M	ATA	130	283
4	??	??	ASY	138	??
5	54	M	??	150	195

A tabela anterior é um recorte com dados sobre a saúde de pacientes, mais especificamente sobre eventuais problemas cardíacos. Não entraremos em detalhes sobre o que cada coluna significa, mas repare que, entre os valores do tipo `string` e valores do tipo numérico, há um tipo de valor que destoa dos demais, que são os símbolos de interrogação (??). Eles representam um dado que não foi coletado. A falta desses dados pode provocar uma análise equivocada ou até mesmo errada sobre a saúde dos pacientes.

Quando encontramos valores ausentes em nosso conjunto de dados, devemos encará-los como um alerta sobre um possível erro ou falha na coleta de dados e devemos corrigir esse problema para que não tenhamos mais valores ausentes. Para isso, temos algumas soluções, por exemplo:

1. Coletar novamente os dados para preencher os valores ausentes com os valores reais;
2. Utilizar técnicas para preencher os valores ausentes por algum outro valor que não cause tanto ruído na nossa base de dados.

Sobre a primeira opção, nem sempre é viável ou possível coletar os dados novamente. Por exemplo, considere a base de dados do censo demográfico brasileiro: em caso de haver valores ausentes, deveríamos localizar novamente a pessoa e fazer as perguntas que

faltam na base de dados? Imagine o custo dessa operação, ou ainda, de praticamente refazer o censo, dependendo da quantidade de valores ausentes. Então, a segunda opção, a de utilizar técnicas para substituir os valores ausentes, seria a mais viável para nós.

A substituição de valores ausentes assume que as ausências implicam perda de informação relevante. Olhando para a tabela anterior novamente, conseguimos dizer o que está acontecendo com esses pacientes nos atributos que estão faltando? O fato de não conseguirmos explicar esses valores ou dizer o que está acontecendo pode implicar omissão de socorro para esses pacientes devido a equívocos ou erros em nossas análises.

Então, quando temos valores ausentes em nossa base de dados, devemos fazer algumas perguntas, como:

1. Qual o motivo de haver valores ausentes dentro deste conjunto?
2. Os dados ausentes pertencem a um conjunto de dados numérico ou categórico? A depender do tipo do nosso conjunto, utilizaremos uma estratégia ou outra para trabalharmos com valores ausentes.

CONJUNTOS NUMÉRICOS E CONJUNTOS CATEGÓRICOS

Conjunto numérico: qualquer conjunto de dados que possua apenas valores numéricos, sendo eles do tipo `int` ou do tipo `float`.

Conjunto categórico: tradicionalmente, é qualquer conjunto que defina uma categoria ou classe, por exemplo, gênero, cidadania, profissão, entre outros. Em Python, assim como em outras linguagens, consideramos como categórico qualquer valor em forma de texto, ou seja, um valor do tipo `string`.

Sabendo a importância de lidar com valores ausentes em nosso conjunto de dados, vamos começar estudando algumas técnicas

para trabalharmos com esse tipo de valor. Primeiro, veremos métodos de substituição de valores ausentes em um conjunto de valores numéricos e, depois, veremos um método de substituição de valores ausentes em conjuntos de dados categóricos.

1.1 Substituição de valores ausentes por valores numéricos

Aplicar a substituição de valores ausentes significa dizer que vamos selecionar os valores ausentes de nossa base de dados e utilizar técnicas para substituí-los por algum outro valor que não provoque grandes alterações em nossa base de dados. Na Estatística, essas alterações são chamadas de ruído.

Ruídos são valores que podem prejudicar a qualidade dos dados. Alguns exemplos de ruído que podemos citar são: valores do tipo *outliers*, valores não coletados ou coletados de forma errada. Esses erros na coleta e no tratamento dos dados promovem falhas no ajuste do modelo de *machine learning*.

Para escolhermos um critério de substituição adequado, é essencial entender a distribuição dos valores ausentes em nosso conjunto de dados. Esses valores podem ser classificados como:

1. Valores ausentes completamente aleatórios;
2. Valores ausentes aleatórios.

Valores ausentes completamente aleatórios

Quando temos *valores ausentes completamente aleatórios*, significa que a ausência de dados não está relacionada a nenhuma característica observável ou não observável dos participanetes. Em outras palavras, todos os dados têm a mesma probabilidade de estar ausentes, independentemente de qualquer fator.

Por exemplo, suponha que foi realizado uma pesquisa com 100 pessoas aleatórias e percebemos que a proporção de pessoas que não respondem à pesquisa é a mesma entre homens e mulheres. Isso caracteriza valores ausentes completamente aleatórios, pois a ausência de resposta não está associada ao gênero ou a qualquer outra característica.

Esse tipo de dados ausentes é preferível, pois não introduz viés na análise. Quando lidamos com valores ausentes completamente aleatórios, podemos utilizar o conjunto de dados como um todo para inferir os valores faltantes sem comprometer a imparcialidade. Alguns exemplos incluem:

- Uma pesquisa em que algumas pessoas pulam perguntas de maneira aleatória
- Um conjunto de dados acadêmicos em que notas estão faltando devido ao não envio das atividades pelos alunos, sem nenhum padrão aparente.

Valores ausentes aleatórios

Os valores ausentes aleatórios ocorrem quando a ausência de dados está associada a características observáveis. Isso significa que a probabilidade de um valor estar usente depende de outros fatores no conjunto de dados.

Por exemplo, suponha que repetimos a pesquisa mencionada anteriormente, mas, dessa vez, notamos que 70% dos homens entrevistados não responderam, enquanto entre as mulheres a taxa de não resposta foi menor. Aqui, a ausência de dados está associada ao sexo, tornando-os valores ausnetes aleatórios.

Embora menos desejáveis do que os *valores completamente aleatórios*, esses dados ainda podem ser úteis. Podemos usar as informações disponíveis para tratar as ausências de forma mais informada e mitigar possíveis vieses. Exemplos comuns incluem:

- Pesquisas de satisfação em que clientes insatisfeitos têm maior probabilidade de não responder a certas perguntas;
- Prontuários médicos com exames ausentes devido a fatores como idade, sexo ou condições de saúde do paciente

Compreender essa diferença nos ajuda a decidir como lidar com valores ausentes de maneira mais eficaz, garantindo análises mais robustas e imparciais.

Na tabela a seguir, temos registrados o gênero e a altura de dez pessoas. Entretanto, alguns valores estão ausentes (`NaN`), e isso pode ocorrer por vários motivos, mas vamos considerar aqui que as pessoas simplesmente pularam essa questão, o que provocou o aparecimento de dados ausentes.

Pessoa	Gênero	Altura
Pessoa 1	Masculino	1,75
Pessoa 2	Masculino	NaN
Pessoa 3	Feminino	1,68
Pessoa 4	Feminino	NaN
Pessoa 5	Feminino	1,71
Pessoa 6	Masculino	NaN
Pessoa 7	Feminino	1,71
Pessoa 8	Feminino	NaN
Pessoa 9	Masculino	1,76
Pessoa 10	Masculino	NaN

O que podemos fazer para corrigir isso é utilizar o nosso conjunto de dados inteiro, ou seja, utilizar os valores que são diferentes de `NaN` , para inferir possíveis valores para esses dados ausentes. Uma das

técnicas que podemos utilizar é calcularmos a média dos nossos valores.

Para calcularmos a média das alturas, utilizamos o seguinte código em Python:

```
def soma_desconsiderando_nulos(vetor):
    soma = 0
    for valor in vetor:
        if valor is not None:
            soma += valor
    return soma
```

Essa função realiza o seguinte:

1. `def soma_desconsiderando_nulos(vetor)` : define uma função chamada `soma_desconsiderando_nulos` , que recebe um argumento chamado `vetor` ;
2. `soma = 0` : inicializa uma variável `soma` com valor zero. Essa variável será usada para acumular a soma dos valores não nulos no vetor.
3. `for valor in vetor:` : esse loop `for` percorre cada elemento do vetor;
4. `if valor is not None:` : dentro do loop, verificamos se o valor atual não é nulo (ou seja, diferente de `None`);
5. `soma += valor` : se o valor não for nulo, ele é adicionado à variável `soma` ;
6. `return soma` : retorna o valor total da soma dos elementos não nulos do vetor.

Veja que utilizamos somente as cinco linhas que possuem valores diferentes de `NaN` . Para este nosso exemplo, podemos inferir que a altura das pessoas que estão faltando é, em média, `1,71` .

Para aprofundarmos no conceito de valores ausentes aleatórios, vejamos agora um exemplo prático. Considere a tabela a seguir os registros de uma pesquisa de satisfação:

Nome	Gênero	Satisfação
Allan	Masculino	6
Thiago	Masculino	7
Alice	Feminino	9
Victoria	Feminino	NaN
Gabriely	Feminino	NaN
Alexandre	Masculino	5
Ana	Feminino	6
Maria	Feminino	NaN
Lucas	Masculino	10
André	Masculino	NaN

Completar os dados faltantes nesse caso é um pouco mais complicado, pois precisaremos fazer mais perguntas ou ter mais dados para conseguirmos utilizar métodos estatísticos para fazer essa inferência no quesito "satisfação". Se fizermos a substituição pela média aqui, substituiremos os valores do tipo `NaN` por `7.3`, o que pode não condizer com a realidade, pois a ausência desse dado pode ter sido provocada pela insatisfação do cliente com o produto ou com o atendimento, ou pelo simples fato de que a pessoa ficou constrangida em dar uma nota baixa e preferiu não responder.

Precisaríamos investigar melhor e procurar evidências que corroborem com a tese de que as pessoas ficaram constrangidas e preferiram não responder e, caso confirmássemos essa tese, poderíamos substituir por uma nota baixa — no caso, `0` — e obter o seguinte dataframe:

Nome	Gênero	Satisfação
Allan	Masculino	6
Thiago	Masculino	7
Alice	Feminino	9
Victoria	Feminino	0
Gabriely	Feminino	0
Alexandre	Masculino	5
Ana	Feminino	6
Maria	Feminino	0
Lucas	Masculino	10
André	Masculino	0

Eu sei que pode ser frustrante, mas análise de dados e engenharia de recursos não são mágica. Nem sempre você conseguirá arrumar os seus dados ou extrair informações novas para ajustar os dados faltantes como no exemplo anterior. O que nos resta é construir um bom planejamento para evitar que esses casos aconteçam.

Após verificarmos a distribuição dos dados ausentes, podemos escolher um critério para substituição. A escolha do valor que utilizaremos para substituir os valores ausentes não deve ser captada como um sinal forte pelos algoritmos de *machine learning*.

SINAL FRACO E SINAL FORTE

Um sinal fraco é aquele que não soma nem subtrai informação do nosso conjunto de dados. Em outras palavras, não causa enviesamento em nossa base de dados.

Já um sinal forte é aquele que causa enviesamento em nossa base de dados. Por exemplo, observe o conjunto numérico `[1,2,3,4,5,NaN]`. Ao executarmos o cálculo da média nesse conjunto, obtemos o valor 3. Se efetuarmos a substituição do valor ausente pelo número 99 e calcularmos a média novamente com esse novo conjunto, `[1,2,3,4,5,99]`, teremos que o número calculado é de 19. Ou seja, o valor 99 foi captado como um sinal forte.

No caso de valores ausentes, desejamos que o algoritmo de *machine learning* ignore os valores inseridos. Para isso, nossa escolha deverá ser a de um número que seja o mais indefinido possível, que não provoque grandes alterações em nosso conjunto de dados.

No caso de valores numéricos, podemos pensar na substituição dos valores ausentes pelo valor médio ou pela mediana do conjunto. Já para valores categóricos, podemos pensar na substituição dos valores ausentes pela categoria frequente do conjunto, também conhecida como moda.

Com isso, podemos ter em mente que dois fatores nos dirão qual método de substituição devemos usar:

- O tipo do conjunto de dados com o qual estamos trabalhando;
- O modelo de *machine learning* que vamos utilizar.

MODELO DE *MACHINE LEARNING*

Um modelo de *machine learning* é um modelo matemático que utilizamos para fazer previsões. Considere que um taxista cobra R$ 5,00 pelo quilômetro rodado, mais R$ 2,50 pela bandeira. Desta forma, poderíamos ter a seguinte função para calcular o valor total da corrida de táxi: `5 * quilômetro + 2,50`. Esse é o nosso modelo matemático. Para prevermos o valor da corrida, poderíamos inserir os locais de origem e destino e, em seguida, calcular a distância em quilômetros para inserir na nossa função. Claro que esse é um exemplo bem simples, mas nem sempre um modelo de *machine learning* é tão simples assim ou consegue predizer algo com exatidão em todos os casos.

Agora que temos uma noção do que vamos fazer, começaremos com a substituição de valores ausentes em um conjunto de valores numéricos e, em seguida, veremos como fazer a substituição em um conjunto de valores categóricos.

Substituição pela média

O primeiro método de substituição que veremos é o método de substituição pela média. Esse método consiste em calcular a média do nosso conjunto de dados e então substituir os valores ausentes por esse valor calculado. Para entender como esse método funciona, vamos considerar a seguinte lista de valores:

```
[10,NaN,1,9,NaN,20,NaN,14,2,20]
```

Vamos substituir os valores ausentes desse conjunto primeiro de forma manual. O primeiro passo é calcular a média. Para calcularmos a média desse nosso conjunto, precisamos desconsiderar os valores ausentes, ou seja, considerar somente os valores diferentes de `NaN`. Veja que, dos 10 valores em nosso conjunto, 3 são valores `NaN`. Para isso, vamos somar todas os valores que não são `NaN`, da seguinte forma:

```
soma = 10+1+9+20+14+2+20
soma = 76
```

Agora vamos dividir o resultado dessa `soma`, 76, pelo comprimento do nosso vetor sem os valores nulos — nesse caso, por 7 (10 valores iniciais menos os 3 valores `NaN`):

```
media = 76/7
media = 10.8571
```

O que fizemos foi dividir essa fórmula em duas partes. Na primeira, somamos os valores diferentes de `NaN` e os armazenamos na variável `soma`. Já na segunda parte, fizemos a divisão da variável `soma`, que tem o valor `76`, pelo comprimento do vetor desconsiderando os valores nulos, que tem o valor `7`. Dessa forma obtivemos o resultado de `10.8571`

Agora que possuímos o valor da média, podemos substituir os valores ausentes (`NaN`) por esse valor calculado. Fazendo isso, nosso conjunto ficará assim:

```
[10,10.8571,1,9,10.8571,20,10.8571,14,2,20]
```

Vamos fazer o cálculo da média novamente, mas agora utilizando os valores substituídos pela média calculada anteriormente:

```
media = (10+10.8571+1+9+10.8571+20+10.8571+14+2+20)/10
media = 10.8571
```

Agora que entendemos como a substituição pela média funciona, suponha que estejamos analisando os dados de uma pesquisa realizada em uma loja com o objetivo de calcular a média de idade dos clientes para determinar a estratégia de direcionamento da campanha publicitária. Devido ao alto volume de clientes que essa loja recebe, de aproximadamente 100 por dia (totalizando 3 mil clientes em 30 dias), calcular manualmente as idades de todos os clientes se torna extremamente difícil e demorado.

No entanto, podemos contar com a linguagem de programação Python para escrever uma função que calcule a média de um conjunto de dados mais de uma vez e que não dependa do número de dados inseridos para realizar o cálculo.

Antes de começarmos a escrever nossa função, a qual chamaremos de `troca_media`, o primeiro passo é importar as bibliotecas que vamos utilizar. Importaremos a biblioteca *pandas*, que utilizaremos para transformar em um dataframe os valores que vamos declarar. Também importaremos a biblioteca *numpy*, que utilizaremos mais adiante para acessar alguns valores especiais, por exemplo, o valor ausente `np.nan`.

> **PANDAS DATAFRAME** é uma estrutura de dados bidimensional com os dados alinhados em forma de tabela, ou seja, dispostos em linhas e colunas. Uma forma de conseguir visualizar um dataframe é pensar nele como uma planilha de Excel.

```
import pandas as pd
import numpy as np

def troca_media(dataframe,coluna):
    media = dataframe[coluna].mean()
    valores_trocados = dataframe[coluna].fillna(media)

    return valores_trocados
```

Nessa função, nós calculamos a média da coluna do nosso dataframe com o comando `mean`. Ao utilizarmos o comando `dataframe[coluna].mean()`, estamos selecionando uma coluna em específico do nosso dataframe e calculando a `media`. Lembra que importamos o `numpy`? Ele retira automaticamente os valores `NaN` para calcular os valores da média. Utilizamos o comando `fillna` passando como parâmetro a `media` para que os valores iguais a `NaN` sejam substituídos pela média.

Com essa função criada, vamos recriar a tabela que utilizamos, porém como um dataframe. Vamos aproveitar e, em vez de substituirmos a coluna inteira, vamos criar uma outra coluna chamada `valores_trocados`, para conseguirmos comparar o antes e o depois da nossa substituição:

```
vetor = [10,np.nan,1,9,np.nan,20,np.nan,14,2,20]
dataframe = pd.DataFrame({"valores":vetor})
dataframe["valores_trocados"] = troca_media(dataframe, "valores")
```

Com as alterações feitas, agora podemos comparar o antes e o depois da substituição com o método `.describe()` da biblioteca *pandas*:

```
dataframe.describe()
```

Esse método `.describe()` retorna um dataframe com duas colunas, e cada coluna é referente a uma coluna do nosso dataframe. No nosso caso, temos em nosso método `.describe()` uma coluna referente aos `valores` e outra referente aos `valores_trocados`. As linhas de retorno do método `.describe()` descrevem algum valor que foi calculado do nosso dataframe:

index	valores	valores_trocados
count	7.0	10.0
mean	10.8571	10.8571
std	7.7120	6.2968
min	1.0	1.0
25%	5.5	9.25
50%	10.0	10.8571
75%	17.0	13.2142
max	20.0	20.0

Vamos entender o que cada uma dessas linhas significa:

- **count**: a contagem de linhas da nossa coluna desconsiderando os dados ausentes;
- **mean**: a média da coluna do nosso conjunto de dados;
- **std**: o desvio padrão do nosso conjunto de dados;
- **min**: o menor valor do nosso conjunto de dados;
- **25%**: o valor do primeiro quartil. O valor que aparece no método `.describe` indica que antes deste valor concentram-se 25% dos nosso dados;
- **50%**: o segundo quartil. O valor que aparece no método `.describe` indica que antes deste valor encontram-se 50% dos nossos dados;
- **75%**: o terceiro quartil. O valor que aparece no método `.describe` indica que antes deste valor encontram-se 75% dos nossos dados;
- **max**: o maior valor do conjunto de dados.

Agora que sabemos o que cada uma das linhas do nosso método `.describe()` significa, podemos comparar o antes e o depois da nossa substituição. Veja a tabela a seguir:

index	valores	valores_trocados	valores_trocados - valores
count	7.0	10.0	3.0
mean	10.8571	10.8571	0.0
std	7.7120	6.2968	-1.4151
min	1.0	1.0	0.0
25%	5.5	9.25	3.75
50%	10.0	10.8571	0.8571
75%	17.0	13.2142	-3.7857

index	valores	valores_trocados	valores_trocados - valores
max	20.0	20.0	0.0

A tabela anterior exibe os valores do método `.describe()` e o cálculo da diferença entre esses valores. Note que, quando a diferença entre o antes e o depois é zero, significa que não teve alteração de valor; quando a diferença é positiva, significa que o valor foi alterado para mais e, quando a diferença é negativa, significa que o valor foi alterado para menos.

> **Alterações no conjunto de dados**
>
> Tenha em mente que qualquer alteração que fizermos em um conjunto de dados provocará alguma distorção. Claro que algumas alterações distorcem mais e outras menos, o que ditará o quanto será distorcido é o seu conjunto de dados.

Substituição pela mediana

O próximo método que veremos é o da substituição de valores ausentes pela mediana. Da mesma forma como fizemos com o método anterior, faremos primeiro de forma manual, com um conjunto pequeno de dados, e então faremos via código. Para este exemplo, considere a seguinte lista:

```
[10,NaN,1,9,NaN,20,NaN,14,2,20]
```

Para calcularmos a mediana desse conjunto de dados, devemos seguir estes passos:

1. Ordenar os valores em ordem crescente, ou seja, do menor para o maior;

2. Contar o tamanho do nosso conjunto de dados. Caso seja ímpar, a mediana do nosso conjunto é o valor que divide o nosso conjunto de dados ao meio. Caso seja par, devemos somar os dois valores do meio e dividir por 2; o resultado será a nossa mediana.

Para ilustrar o cálculo da mediana, o primeiro passo é ordenar os valores do nosso conjunto em ordem crescente:

```
[1,2,9,10,14,20,20,NaN,NaN,NaN]
```

Os valores ausentes foram agrupados no final da nossa tabela, mas isso não faz diferença no resultado, pois eles serão desconsiderados em nosso cálculo da mediana. Agora vamos localizar o valor que divide nosso conjunto de dados ao meio. No nosso caso, temos 7 valores que não são ausentes, portanto, uma quantidade ímpar. Então, precisamos achar em nosso conjunto o valor que o divide em 3 valores à esquerda e 3 valores à direita deste valor que chamamos de mediana. Sendo assim, o valor que encontramos como a mediana do nosso conjunto é o que ocupa a 4ª posição, o valor `10`.

Agora podemos substituir os valores ausentes do nosso conjunto de dados pelo valor `10`, ficando com a seguinte lista:

```
[1,2,9,10,14,20,20,10,10,10]
```

Vamos recalcular a nossa mediana. Para isso, vamos seguir os passos 1 e 2 novamente. Vamos ordenar nosso conjunto novamente em ordem crescente:

```
[1,2,9,10,10,10,10,14,20,20]
```

Se tentarmos localizar o valor do meio agora, percebemos que ele não existe, pois agora temos uma quantia par de valores. Então, vamos usar a outra estratégia para calcularmos a mediana dos nossos dados: escolher dois valores de tal forma que consigamos dividir nosso conjunto ao meio. Sendo assim, os valores que escolheremos são os valores que ocupam a 5ª e a 6ª posição: ficamos com 4 valores antes da 5ª posição e 4 valores depois da 6ª.

Agora, para calcular a mediana, vamos somar os dois valores selecionados e dividir por 2:

```
mediana = (10+10)/2
mediana = 20/2
mediana = 10
```

Ou seja, da mesma forma como a média, o valor da mediana não foi alterado após a substituição dos valores ausentes do nosso conjunto de dados. Vamos fazer essa mesma substituição, porém via código. Repare que o código que vamos utilizar para fazer essa substituição é muito parecido com o que utilizamos para fazer a substituição pela média. A única linha que sofre uma alteração significativa é a quinta linha, onde, em vez de utilizarmos o comando `mean` do `numpy` para calcular a média da coluna, utilizamos o comando `median` para calcular a mediana.

```
import pandas as pd
import numpy as np

def troca_mediana(dataframe,coluna):
    mediana = dataframe[coluna].median()
    valores_trocados = dataframe[coluna].fillna(mediana)

    return valores_trocados
```

Agora que escrevemos a função para fazer a substituição pela mediana, vamos utilizá-la em um dataframe para ver o seu funcionamento. Para isso, vamos recriar a tabela que utilizamos como exemplo, agora em formato de dataframe.

```
vetor = [10,np.nan,1,9,np.nan,20,np.nan,14,2,20]

dicionario = {"valores":vetor}
dataframe = pd.DataFrame(dicionario2)
dataframe["valores_trocados"] = troca_mediana(dataframe,"valores")
```

Da mesma forma como na substituição pela média, em vez de sobrescrevermos a coluna, nós criamos outra coluna para podermos

comparar o antes e o depois com o comando `describe`, conforme mostra a tabela a seguir:

	valores	valores_trocados
count	7.000000	10.000000
mean	10.857143	10.600000
std	7.712081	6.310485
min	1.000000	1.000000
25%	5.500000	9.250000
50%	10.000000	10.000000
75%	17.000000	13.000000
max	20.000000	20.000000

Como já conhecemos cada linha do comando `describe`, vamos à tabela de comparação entre as duas colunas:

	valores	valores_trocados	valores_trocados - valores
count	7.00	10.00	3.00
mean	10.8571	10.60	-0.2571
std	7.7120	6.3104	-1.4015
min	1.00	1.00	0.00
25%	5.50	9.250	3.75
50%	10.00	10.00	0.00
75%	17.00	13.00	-4.00
max	20.00	20.00	0.00

A terceira coluna indica a diferença entre a coluna original de valores e a de valores trocados. Podemos perceber que a nossa mediana não foi alterada, porém os outros valores, como a média, o desvio padrão, o primeiro quartil (25%) e o terceiro quartil (75%) foram alterados nessa substituição de valores ausentes.

Como já afirmamos, o que foi apresentado na tabela não é uma regra: você terá que testar as substituições e ver qual delas enviesa menos os seus dados. Aqui, conseguimos ver que a substituição pela mediana também provoca distorções em nossos dados, mesmo que pequenas.

1.2 Substituição de valores ausentes em conjuntos de dados categóricos

Até o presente momento, vimos técnicas de substituição de valores ausentes para conjuntos de dados numéricos. Porém, como devemos proceder quando nosso conjunto de dados não for numérico, mas categórico? Veja que, para esse tipo de conjunto de dados, não temos como calcular a média ou a mediana, então precisamos de alguma outra estratégia para a substituição.

Uma estratégia que podemos utilizar é substituir valores ausentes com o valor de maior frequência, também conhecida como substituição pela moda. Da mesma forma como fizemos com as outras substituições, nesta também começaremos pelo processo manual e, em seguida, partiremos para o código. Para o nosso exemplo, considere a tabela a seguir:

Sobrenome	Idade	Gênero
Silva	71	Masculino
Cavalcanti	35	NaN
Oliveira	61	Feminino
Santos	73	Masculino
Souza	28	Feminino
Lima	41	NaN
Pereira	35	Feminino
Rodrigues	70	NaN
Carvalho	46	Masculino
Alves	68	Feminino

A tabela que acabamos de ver contém o sobrenome, a idade e o gênero da pessoa; porém, note que, na coluna `gênero`, temos três valores ausentes, ou seja, três valores iguais a `NaN`. Caso tivéssemos o nome das pessoas, poderíamos criar um algoritmo de processamento de linguagem natural para identificarmos o possível gênero da pessoa pelo nome, mas, como não é o caso, teremos que substituir os valores ausentes utilizando a moda.

Para isso, devemos seguir alguns passos:

1. Contar quantas categorias aparecem em nosso conjunto de dados;
2. Contar quantas vezes cada categoria aparece em nosso conjunto de dados;
3. Selecionar a categoria com a maior incidência em nosso dataframe e utilizá-la para substituir pelos valores ausentes.

Em nossa coluna de gênero, temos apenas duas categorias: `Masculino` e `Feminino`, sendo que a categoria `Masculino` aparece três vezes e a categoria `Feminino` aparece quatro vezes. Desta forma, o valor de categoria frequente que utilizaremos para substituir os valores ausentes será a categoria `Feminino`.

Substituindo os valores ausentes pela categoria `Feminino`, temos a seguinte tabela:

Sobrenome	Idade	Gênero
Silva	71	Masculino
Cavalcanti	35	Feminino
Oliveira	61	Feminino
Santos	73	Masculino
Souza	28	Feminino
Lima	41	Feminino
Pereira	35	Feminino
Rodrigues	70	Feminino
Carvalho	46	Masculino
Alves	68	Feminino

Da mesma forma como fizemos com os valores numéricos, podemos contar a frequência das categorias `Feminino` e `Masculino` novamente:

Categoria	Contagem
Feminino	7
Masculino	3

Assim, vimos que a moda não foi alterada mesmo com a substituição dos valores ausentes. Antes de continuarmos, quero que preste

atenção na seguinte tabela:

Sobrenome	Idade	Gênero
Silva	71	M
Cavalcanti	35	F
Oliveira	61	Feminino
Santos	73	masculino
Souza	28	feminino

Quantas categorias temos na tabela apresentada? À primeira vista, é simples dizer que ainda temos duas categorias: a categoria "Feminino", que na tabela está sendo representada pelos valores `F`, `Feminino` e `feminino`; e a categoria "Masculino", que na tabela está sendo representada pelos valores `M` e `masculino`. Entretanto, para o computador isso não é tão simples assim. Para a biblioteca *pandas*, por exemplo, a tabela anterior possui cinco categorias, que estão descritas na tabela a seguir:

Categoria	Contagem
feminino	1
Feminino	1
masculino	1
F	1
M	1

A falta de padronização de dados categóricos pode ser um problema não tão raro de se acontecer no dia a dia. Quando estamos trabalhando com um conjunto de dados categóricos, devemos antes prestar atenção nas categorias e se elas estão padronizadas; caso não estejam, podemos ter problemas na substituição.

No caso de valores categóricos, temos como solução para substituição de valores ausentes a moda. Embora seja muito utilizado em algumas situações, esse método de substituição não é aconselhável, pois poderíamos fazer algum tipo de imputação que não é verdadeira, o que nos levaria a um equívoco ou erro de análise.

Vamos fazer essa mesma substituição via código e você verá que o código para substituição de valores ausentes pela moda é semelhante aos demais códigos que fizemos, sendo que a diferença mais visível está na quarta linha, onde, em vez de utilizarmos o comando `mean` ou `median` da biblioteca *pandas*, utilizaremos o comando `mode()[0]`.

Utilizamos o comando `mode()[0]` porque, em alguns casos, o nosso conjunto de dados possui mais de um valor frequente (ou moda) e, por causa disso, o comando `mode()` retorna uma lista com todos os valores frequentes. Devemos escolher algum valor desta lista, então aqui utilizamos a notação `[0]` para capturarmos o primeiro valor desta lista.

```
import pandas as pd
import numpy as np

def cat_frequente(dataframe,coluna):
    moda = dataframe[coluna].mode()[0]
    valores_trocados = dataframe[coluna].fillna(moda)

    return valores_trocados
```

Para utilizar a função que acabamos de criar, usaremos um dataframe semelhante à tabela do nosso exemplo:

```
    dicionario = {"sobrenome":
["Silva","Cavalcanti","Oliveira","Santos","Souza","Lima","Pereira","Rodrig
ues","Carvalho","Alves"],
        "idade":[71,35,61,73,28,41,35,70,46,68],
        "genero":
["Masculino",np.nan,"Feminino","Masculino","Feminino",np.nan,"Feminino",np
```

```
.nan,"Masculino","Feminino"]}

    dataframe = pd.DataFrame(dicionario)
    dataframe["genero_alterado"] = cat_frequente(dataframe, "coluna")
```

Desta forma, conseguimos fazer a substituição de valores ausentes pela moda.

Neste capítulo, vimos como valores ausentes podem ser problemáticos em nossa base de dados e como devemos proceder quando encontrarmos algum valor do tipo ausente em nosso conjunto de dados. Vimos primeiramente como substituir valores ausentes pela média ou pela mediana em um conjunto de dados numéricos. Fizemos alguns testes para analisar o enviesamento provocado por essas substituições e terminamos fazendo a substituição dos valores ausentes pela moda em um conjunto de valores categóricos.

CAPÍTULO 2
Projeto prático — Substituição de valores ausentes

Até o presente momento aprendemos a fazer a substituição de valores ausentes utilizando a biblioteca pandas e nenhuma outra ferramenta para fazer a automação deste processo para nós. Entretanto, a atividade de fazer substituição de valores ausentes é uma tarefa comum no dia a dia da pessoa cientista de dados e já existem ferramentas prontas para fazermos essas substituições. Neste projeto prático, utilizaremos as bibliotecas *scikit-learn* e a *feature-engine* para nos auxiliarem com a substituição de valores ausentes.

2.1 Conhecendo a nossa base de dados

O primeiro projeto prático explorará um conjunto de dados bem conhecido entre as pessoas que estão começando a estudar Ciência de Dados: o Titanic. O desafio está em conseguir criar um algoritmo de *machine learning* capaz de prever as pessoas que sobreviveram ao desastre, porém, neste primeiro momento, trabalharemos apenas com a imputação de valores ausentes.

> Os dados do Titanic e o notebook com os códigos desta parte podem ser encontrados no meu repositório do GitHub. Acesse: https://github.com/ThiagoBenevides/feature-engine-transformando-dados-brutos-em-recursos/tree/main/capitulo_dois

Antes de submetermos os nossos dados ao algoritmo de *machine learning*, precisamos fazer o que é chamado de **pré-**

processamento de dados, isto é, precisamos deixar os dados padronizados para serem consumidos pelos modelos de *machine learning*.

O primeiro passo quando estamos diante de uma base de dados é conhecê-la. Para isso, podemos nos fazer as seguintes perguntas:

1. Quantas colunas e quantas linhas nosso conjunto de dados possui?
2. O que cada coluna deste dataframe significa?
3. Ele possui valores ausentes? Se sim, qual é a porcentagem do nosso dataframe que está ausente?
4. Os dados ausentes são numéricos ou categóricos?
5. Qual será nosso objetivo? Estamos trabalhando com esse dataframe para responder a quais perguntas?

Ao conseguirmos responder a essas perguntas, poderemos traçar um plano de ação para trabalhar com o nosso conjunto de dados. Vamos responder a cada uma delas.

1. Quantas colunas e quantas linhas nosso conjunto de dados do Titanic possui?

Para respondermos a essa pergunta, devemos carregar o conjunto de dados com a biblioteca `pandas`, da seguinte forma:

```
import pandas as pd


dataframe = pd.read_csv("titanic.csv")
dataframe.head(5)
```

Com o código anterior, fizemos a importação do conjunto de dados do Titanic e imprimimos as 5 primeiras linhas. O resultado pode ser visto na tabela a seguir — por uma questão visual, ela está dividida em duas partes:

PassengerId	Survived	Pclass	Name	Sex	Age
1	0	3	Mr. Owen Harris	male	22.0
2	1	1	John Bradley	female	38.0
3	1	3	Miss. Laina	female	26.0
4	1	1	Mrs. Jacques Heath	female	35.0
5	0	3	Mr. William Henry	male	35.0

PassengerId	Parch	Ticket	Fare	Cabin	Embarl
1	0	A5 21171	7.25	NaN	S
2	0	PC 17599	71.2833	C85	C
3	0	STONO2. 3101282	7.925	NaN	S
4	0	113803	53.1	C123	S
5	0	373450	8.05	NaN	S

E agora podemos utilizar o método `.shape` da seguinte forma: `dataframe.shape()`. O resultado deste código será uma tupla com os seguintes valores: `(891,12)`. O primeiro número, 891, faz referência ao número de linhas do nosso dataframe e o número 12 diz respeito ao número de colunas. Sendo assim, nosso dataframe com os dados do Titanic tem 891 linhas e 12 colunas.

2. O que cada coluna deste dataframe significa?

Saber o que cada coluna significa é importante, pois vai ajudar a dizer o que queremos utilizar como variável-alvo (o que queremos prever deste dataframe) e as variáveis que são preditoras (as variáveis que desejaremos utilizar para prever nossa variável-alvo). Esse conhecimento também auxiliará na detecção de erros em nosso dataframe. Por exemplo, se estivermos trabalhando com uma coluna de idades, sabemos que, caso exista ali uma idade negativa ou um número elevado (por exemplo, 999), existe algum erro neste valor, então devemos tratá-lo.

Vamos ver o que cada uma das colunas do nosso conjunto do Titanic significa:

- `PassengerId` : identificador único do(a) passageiro(a);
- `Survived` : indica se o(a) passageiro(a) sobreviveu ao desastre ou não, sendo `0` para não e `1` para sim;
- `Pclass` : classe do ingresso. Apresenta também um indicador socioeconômico, sendo `1` para alto, `2` para médio e `3` para baixo;
- `Name` : nome do(a) passageiro(a);
- `Sex` : gênero do(a) passageiro(a). Nesse conjunto de dados, tratamos apenas a divisão binária de gênero, ou seja, feminino e masculino;
- `Age` : idade do(a) passageiro(a), sendo que em algumas linhas possuímos uma idade fracionada, por exemplo, 23.5. Quando isso ocorre, é porque a idade foi estimada;
- `SibSp` : representa o número de cônjuges a bordo para cada passageiro(a), o que inclui irmãos e irmãs, meios-irmãos ou irmãs, maridos e esposas. Por exemplo, se um passageiro tiver dois irmãos e uma esposa a bordo, o valor dessa coluna será de 3;
- `Parch` : representa o número de pais e filhos a bordo para cada passageiro(a). Por exemplo, se um passageiro tiver um pai e dois filhos a bordo, o número dessa coluna será de 3.
- `Ticket` : número do bilhete do(a) passageiro(a);
- `Fare` : tarifa do(a) passageiro(a);
- `Cabin` : número da cabine do(a) passageiro(a);
- `Embarked` : apresenta o lugar onde a pessoa embarcou. São três lugares possíveis: Queenstown (`Q`), Southampton (`S`) e Cherbourg (`C`).

Com isso, temos que a nossa coluna-alvo é a `Survived` e todas as outras serão tratadas como variáveis previsoras.

3. Esse dataframe possui valores ausentes?

Para descobrirmos se existem valores ausentes, vamos utilizar dois métodos em conjunto: o primeiro é o `.isnull()` , que cria uma tabela com valores `True` ou `False` (sendo `True` quando o valor for igual a `NaN` e `False` quando não for `NaN`); em seguida, utilizaremos o

comando `sum()` para contabilizarmos quantos valores igual a `True` temos em cada coluna:

```
dataframe.isnull().sum()
```

E o resultado que temos pode ser visto a seguir:

```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex            267
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
```

Podemos ver que algumas colunas do nosso dataframe possuem valores ausentes: a `Sex` possui 267 valores ausentes; a `Age`, 177; a `Cabin`, 687, e a `Embarked`, 2. Além de sabermos a soma dos valores ausentes em nosso dataframe, podemos saber qual a porcentagem dos valores ausentes em relação ao total de linhas. Para isso, em vez de utilizarmos o comando `isnull().sum()`, utilizamos o comando `isnull().mean()` e, deste modo, o nosso retorno serão valores entre 0% e 100%. Sendo assim, temos as seguintes porcentagens de valores ausentes em cada uma das colunas:

- `Sex`: 29.96% de valores faltantes;
- `Age`: 19.86% de valores faltantes;
- `Cabin`: 77.10% de valores faltantes;
- `Embarked`: 0.22% de valores faltantes.

4. Os dados ausentes são numéricos ou categóricos?

Para sabermos se os dados são categóricos ou numéricos, podemos utilizar o comando `dataframe.dtypes` da biblioteca `pandas` e o retorno

será o tipo de valor presente em cada coluna, como listado a seguir:

```
PassengerId      int64
Survived         int64
Pclass           int64
Name            object
Sex             object
Age            float64
SibSp            int64
Parch            int64
Ticket          object
Fare           float64
Cabin           object
Embarked        object
```

Então, os valores que temos em nosso dataframe são :

- Valores do tipo `int64` e `float64`, que representam valores numéricos inteiros e de ponto flutuante, respectivamente;
- Valores do tipo `object`, que são valores do tipo `string`.

Podemos traçar algumas estratégias para trabalhar com os valores ausentes. Por exemplo, nas colunas `Sex` e `Cabin`, podemos fazer a substituição de valores ausentes pela moda; já nas colunas `Age`, `Parch` e `Fare` podemos fazer a substituição de valores ausentes pela média ou mediana e pela moda. Para o nosso projeto prático, veremos como fazer as substituições pela média e pela mediana.

5. Qual será nosso objetivo? Estamos trabalhando com esse dataframe para responder a quais perguntas?

O nosso objeto é prever quais passageiros e passageiras sobreviveram ao desastre.

Agora que respondemos a todas as perguntas, podemos começar a preparar o dataframe para o trabalho de substituição de valores ausentes.

Fase de preparação

Vamos iniciar a fase de preparação do nosso conjunto de dados dividindo-o em um conjunto de treino e outro de teste utilizando a biblioteca *scikit-learn*:

```
from sklearn.model_selection import train_test_split

X_treino, X_teste, y_treino, y_teste =
train_test_split(dataframe.drop("Survived",axis=1),dataframe["Survived"],t
est_size=0.3,random_state=0)
```

Dentro desse método `train_test_split`, passamos os seguintes parâmetros:

- `dataframe.drop("Survived",axis=1)`: nesse parâmetro, estamos excluindo a coluna de nome `Survived` e passando somente as variáveis que utilizaremos para prevê-la;
- `dataframe["Survived"]`: aqui estamos indicando a variável que desejamos prever, ou seja, a variável-alvo;
- `test_size=0.3`: nesse parâmetro, indicamos qual será o tamanho do nosso conjunto de dados. Aqui, estamos indicando que o nosso conjunto de dados terá 30% do tamanho do dataframe original. Isso significa que, se o nosso dataframe tem 418 linhas, o nosso conjunto de teste terá 30% desse valor, ou seja, terá 125 linhas. Consequentemente, nosso conjunto de treino terá 293 linhas;
- `random_state`: parâmetro utilizado para garantir a reprodutibilidade do experimento, pois o `train_test_split` seleciona as linhas do conjunto de treino e de teste de maneira aleatória. Isso significa que, se o `random_state` não estiver indicado, a cada execução do programa teremos um novo conjunto de treino e um novo conjunto de teste, o que pode ocasionar alguma diferença entre os resultados.

Agora vamos analisar o que cada uma das variáveis `X_treino`, `X_teste`, `y_treino` e `y_teste` armazena. Para visualizarmos esses dados, podemos utilizar o comando `X_treino.head()` e `X_teste.head()`, e veremos os dados divididos em conjunto de dados de treino e

teste, porém sem a coluna `Survived`, que selecionamos para armazenar na coluna `y_treino` e `y_teste`.

Vamos analisar o dataframe `X_treino` com o comando `X_treino.head(5)`. Utilizamos esse código para exibir apenas as 5 primeiras linhas do nosso dataframe. O resultado pode ser visto na tabela a seguir (dividida em duas, para melhor visualização dos dados):

PassengerId	Pclass	Name	Sex	Age	SibSp
858	1	Peter Denis	male	51.0	0
53	1	Mrs. Henry Sleeper	female	49.0	1
387	3	Sidney Leonard	male	1.0	5
125	1	Percival Wayland	male	54.0	0
579	3	Joseph	female	NaN	1

PassengerId	Parch	Ticket	Fare	Cabin	Embarke
858	0	113055	26.55	E17	S
53	0	PC 17572	76.7292	D33	C
387	2	CA 2144	46.9	NaN	S
125	1	35281	77.2875	D26	S
579	0	2689	14.4583	NaN	C

Aparentemente nada mudou do nosso dataframe original para o dataframe `X_treino` (com exceção da coluna `Survived`, que retiramos), mas veremos que a diferença está em relação ao número de linhas do nosso dataframe `X_treino`. Se fizermos o comando `X_treino.shape`, teremos como retorno `(623, 11)`, ou seja, o dataframe `X_treino` possui uma quantidade de linhas menor que o dataframe original.

A mesma coisa com o dataframe `X_teste`. Se fizermos `X_teste.shape`, teremos como retorno `(268, 11)`. Isso acontece porque dividimos

nosso dataframe original em dois, tanto é que, se somarmos o número de linhas do dataframe `X_treino` com o do `X_teste`, teremos o número de linhas do nosso dataframe original.

Agora que entendemos o nosso dataframe e o dividimos em dados de treino e dados de teste, podemos prosseguir com as substituições de valores ausentes.

2.2 Substituição dos valores ausentes em conjuntos de dados numéricos

Após preparação dos nossos dados, vamos fazer a substituição de valores ausentes. Primero veremos como fazer essa substituição de valores numéricos utilizando a biblioteca *scikit-learn*, depois veremos como utilizar a biblioteca *feature-engine* e então faremos algumas comparações entre as duas bibliotecas.

Substituindo valores ausentes pela média com a biblioteca scikit-learn

A primeira coisa que faremos é uma outra divisão em nosso conjunto de dados. Vamos separar em uma nova variável apenas os valores com que vamos trabalhar, que são os numéricos. Faremos isso pois o *scikit-learn* fará os cálculos de média e mediana de todo o nosso conjunto de valores numéricos e, caso o nosso conjunto de dados tenha valores categóricos, receberemos um erro.

O código a seguir faz a separação dos dados numéricos em duas variáveis, sendo uma para os dados de treino e outra para os valores de teste:

```
numericos_treino
=X_treino[["PassengerId","Pclass","Age","SibSp","Parch","Fare"]]
numericos_teste
=X_teste[["PassengerId","Pclass","Age","SibSp","Parch","Fare"]]
```

O comando que acabamos de executar cria dois dataframes, `numericos_treino` e outro `numericos_teste`. Podemos utilizar o comando `numericos_treino.head()` para observarmos o nosso dataframe. O resultado da execução desse comando é uma tabela similar a esta:

PassengerId	Pclass	Age	SibSp	Parch	Fare
988	1	76.00	1	0.0	78.8500
1273	3	26.00	0	0.0	7.8792
981	2	2.0	1	1.0	23.0000
1125	3	28.0	0	0.0	7.8792
1083	1	28.0	0	0.0	26.0000

Da mesma forma, podemos exibir os dados do conjunto `numericos_teste` com o comando `numericos_teste.head()`, e o resultado pode ser visto a seguir:

PassengerId	Pclass	Age	SibSp	Parch	Fare
1252	3	14.5	8	2.0	69.5500
1062	3	28.0	0	0.0	7.5500
1116	1	53.0	0	0.0	27.4458
1250	3	28.0	0	0.0	7.7500
1201	3	45.0	1	0.0	14.1083

Faremos a substituição dos valores ausentes nestes dois conjuntos de dados. Para começar a substituição, vamos importar o módulo da biblioteca *scikit-learn* chamado `SimpleImputer`. O método `SimpleImputer` será o método que utilizaremos para fazer o cálculo da média. Grosso modo, ele serve para fazermos a substituição de valores ausentes por algum outro valor que definimos no parâmetro

`strategy`. Nesse caso, vamos definir que desejamos fazer a substituição pela média, `mean`.

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="mean")
```

A fim de treinar o nosso algoritmo — que aqui está representado pela palavra `imputer` —, para que ele se ajuste aos valores das colunas, precisamos usar o método `fit`, como mostra o código a seguir:

```
imputer.fit(numericos_treino)
```

Então, se executarmos o seguinte comando `imputer.statistics_`, teremos como resposta:

```
array([4.47678973e+02, 2.28731942e+00, 2.99153386e+01,5.31300161e-
01,3.93258427e-01, 3.24582726e+01])
```

Veja que o retorno foi um numpy array contendo 6 valores, sendo que cada um desses valores pertence a uma coluna e representa a média das colunas que passamos para o nosso `imputer` com o `imputer.fit`.

> O **NUMPY ARRAY** é um objeto que pode ser unidimensional ou bidimensional. Quando é unidimensional, ele se assemelha a um vetor — por exemplo, `[1,2,3,4]`; quando é bidimensional, ele se assemelha a uma matriz — por exemplo, `[[1,2],[3,4],[5,6]]`.

Agora vamos transformar os dados, isto é, precisamos substituir os valores ausentes por esses valores aprendidos. Para isso, utilizaremos outro método da biblioteca *scikit-learn*: o método `transform`.

```
numericos_treino = imputer.transform(numericos_treino)
numericos_teste = imputer.transform(numericos_teste)
```

O resultado do código anterior é um array bidimensional com os valores transformados. Então, nosso objetivo será transformar esse *array* em um dataframe novamente, e para isso usaremos o seguinte código:

```
numericos_treino = pd.DataFrame(numericos_treino,columns=
["PassengerId","Pclass","Age","SibSp","Parch","Fare"])
numericos_teste = pd.DataFrame(numericos_teste,columns=
["PassengerId","Pclass","Age","SibSp","Parch","Fare"])
```

O código que executamos cria dois dataframes, `numericos_treino` e `numericos_teste`, com as seguintes colunas: `PassengerId`, `Pclass`, `Age`, `SibSp`, `Parch` e `Fare`. Esses dataframes são criados a partir dos valores `numericos_treino` e `numericos_teste`, respectivamente. Os valores de `numericos_treino` são armazenados no dataframe de mesmo nome, ou seja, `numericos_treino`. Da mesma forma, os valores de `numericos_teste` são armazenados no dataframe `numericos_teste`.

Após transformamos novamente nossos conjuntos de treino e de teste em um dataframe, podemos verificar o resultado. Para o nosso dataframe `numericos_treino`, se executarmos o comando `numericos_treino.head()`, teremos como resultado uma tabela similar a esta:

index	PassengerId	Pclass	Age	SibSp	Parch	Fa
0	858.0	1.0	51.0	0.0	0.0	26.
1	53.0	1.0	49.0	1.0	0.0	76.
2	387.0	3.0	1.0	5.0	2.0	46.
3	125.0	1.0	54.0	0.0	1.0	77.
4	579.0	3.0	29.91	1.0	0.0	14.

Da mesma forma, para o nosso conjunto de testes, se executarmos o comando `numericos_teste.head()`, veremos uma tabela similar a esta:

index	PassengerId	Pclass	Age	SibSp	Parch	Fa
0	496.0	3.0	29.91	0.0	0.0	14.
1	649.0	3.0	29.91	0.0	0.0	7.5
2	279.0	3.0	7.0	4.0	1.0	29.
3	32.0	1.0	29.91	1.0	0.0	146
4	256.0	3.0	29.0	0.0	2.0	15.

Agora podemos utilizar os métodos da biblioteca `pandas` novamente. Vamos começar pela soma dos valores ausentes nos conjuntos de treino e de teste. Primeiro, vamos utilizar o comando `numericos_treino.isnull().sum()` para calcular os valores ausentes em nosso conjunto de treino:

```
PassengerId    0
Pclass         0
Age            0
SibSp          0
Parch          0
```

```
Fare           0
dtype: int64
```

Para calcular os valores ausentes do nosso conjunto de teste, usamos o comando `numericos_teste.isnull().sum()` :

```
PassengerId    0
Pclass         0
Age            0
SibSp          0
Parch          0
Fare           0
dtype: int64
```

Como podemos ver, não possuímos mais nenhum valor ausente em nossos dataframes, ou seja, obtivemos êxito em nossa tarefa de substituir os valores ausentes pela média utilizando a biblioteca *scikit-learn*.

Substituindo os valores ausentes pela mediana utilizando a biblioteca scikit-learn

Para substituirmos os valores ausentes pela mediana, utilizaremos o mesmo conjunto de valores da seção passada antes das transformações, ou seja, utilizaremos os dataframes `numericos_treino` e `numericos_teste` . Vale lembrar que, para que você consiga ter sucesso utilizando esse método, seus dataframes precisam conter valores ausentes novamente, conforme mostrado com o comando `numericos_treino.isnull().sum()` :

```
PassengerId      0
Pclass           0
Age            121
SibSp            0
Parch            0
Fare             0
dtype: int64
```

E depois com o comando `numericos_teste.isnull().sum()` :

```
PassengerId     0
Pclass          0
Age            56
SibSp           0
Parch           0
Fare            0
dtype: int64
```

Vamos começar importando da biblioteca *scikit-learn* o método `SimpleImputer` para fazermos a substituição dos valores ausentes pela mediana — o mesmo que utilizamos para fazer a substituição pela média. A única diferença aqui será em relação à estratégia passada no parâmetro `strategy`, que não será `mean`, mas `median`, como mostra o código a seguir:

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
```

Agora podemos executar os passos que já conhecemos: primeiro treinaremos nossos dados e depois os transformaremos, com os comandos `fit` e `transform`, respectivamente:

```
imputer.fit(numericos_treino)
numericos_treino = imputer.transform(numericos_treino)
numericos_teste = imputer.transform(numericos_teste)
```

Da mesma forma como antes, os valores transformados agora são do tipo `numpy array` e não mais um dataframe. Então, precisamos convertê-los novamente em um dataframe, como no exemplo anterior. Para isso, basta executar o seguinte comando:

```
numericos_treino = pd.DataFrame(numericos_treino,columns=
["PassengerId","Pclass","Age","SibSp","Parch","Fare"])
numericos_teste = pd.DataFrame(numericos_teste,columns=
["PassengerId","Pclass","Age","SibSp","Parch","Fare"])
```

E o resultado são, novamente, dois dataframes sem valores ausentes, o que pode ser conferido com os comandos:

- `numericos_treino.isnull.sum()`
- `numericos_teste.isnull().sum()`

Substituição de valores ausentes pela média utilizando a biblioteca feature-engine

Faremos a mesma substituição de valores ausentes pela média, porém agora utilizando outra biblioteca, a *feature-engine*. Diferente da *scikit-learn*, não precisamos nos preocupar em separar os valores numéricos dos categóricos, então podemos utilizar os nossos conjuntos `X_treino` e `X_teste`. Caso você esteja utilizando o *Google Colaboratory*, você precisará instalar a biblioteca *feature-engine* antes de utilizá-la, diferente da *scikit-learn*, que já vem instalada. A instalação da *feature-engine* pode ser feita com o seguinte código:

```
!pip install feature-engine
```

Caso dê tudo certo, você receberá uma mensagem dizendo que a instalação foi feita com sucesso. Agora que instalamos a biblioteca, podemos fazer a importação do módulo que utilizaremos, que é o `MeanMedianImputer`:

```
from feature_engine.imputation import MeanMedianImputer
```

Assim como fizemos para o método `SimpleImputer`, precisamos instanciar o método `MeanMedianImputer` e, de alguma forma, fazer com que ele aprenda as médias das colunas que queremos. O método que utilizaremos para isso é o mesmo da biblioteca *scikit-learn*: o método `.fit`. A principal diferença aqui será na hora de instanciarmos o `MeanMedianImputer`, pois, além de passarmos como estratégia a média, passaremos as variáveis que desejamos alterar:

```
imputer = MeanMedianImputer(imputation_method="mean",variables=
['PassengerId', 'Pclass', 'Age', 'SibSp', 'Parch', 'Fare'])
```

Vamos ajustar o modelo aos dados do dataframe `numericos_treino` que contém valores numéricos faltantes; para isso, utilizaremos o seguinte código:

```
imputer.fit(numericos_treino)
```

E, da mesma forma como fizemos com o *scikit-learn*, podemos inspecionar os valores da média feita pelo nosso `imputer`:

```
imputer.imputer_dict_
```

O resultado desse comando é um dicionário no qual as chaves são os nomes das colunas e os valores são as médias das colunas que passamos no nosso `imputer`, como podemos ver a seguir:

```
{'PassengerId': 447.6789727126806,
 'Pclass': 2.287319422150883,
 'Age': 29.915338645418327,
 'SibSp': 0.5313001605136437,
 'Parch': 0.39325842696629215,
 'Fare': 32.45827255216693}
```

Com isso, conseguimos ver a primeira diferença entre as bibliotecas. Enquanto a biblioteca *scikit-learn* retorna um *array* com o comando `statistics_`, a biblioteca *feature-engine* retorna um dicionário de dados com o método `imputer_dict_`.

Com o modelo treinado, podemos transformá-lo, e para isso utilizaremos o mesmo método que usamos com a biblioteca *scikit-learn*: o método `transform`.

```
numericos_treino = imputer.transform(numericos_treino)
numericos_teste = imputer.transform(numericos_teste)
```

Diferente da biblioteca *scikit-learn*, a *feature-engine* devolve para nós um dataframe; dessa forma, não há necessidade de convertermos os valores como quando utilizamos a *scikit-learn*.

Para os dados de treino, temos o seguinte resultado após a transformação:

index	PassengerId	Pclass	Age	SibSp	Parch	Fa
857	858	1	51.0	0	0	26.
52	53	1	49.0	1	0	76.
386	387	3	1.0	5	2	46.
124	125	1	54.0	0	1	77.
578	579	3	29.91	1	0	14.

E, para os dados de teste, temos:

index	PassengerId	Pclass	Age	SibSp	Parch	Fa
495	496	3	29.91	0	0	14.
648	649	3	29.91	0	0	7.5
278	279	3	7.0	4	1	29.
31	32	1	29.91	1	0	146
255	256	3	29.0	0	2	15.

Agora podemos utilizar o método `isnull().sum()` para calcular os dados ausentes dos nossos conjuntos de treino e de teste. Para o dataframe `numericos_treino`, utilizamos o comando `numericos_treino.isnull().sum()` e obtemos o seguinte retorno:

```
PassengerId    0
Pclass         0
Age            0
SibSp          0
Parch          0
Fare           0
dtype: int64
```

Para o dataframe `numericos_teste`, utilizamos o comando `numericos_teste.isnull.sum()` e temos o seguinte resultado:

```
PassengerId    0
Pclass         0
Age            0
SibSp          0
Parch          0
Fare           0
dtype: int64
```

Conseguimos novamente substituir os valores ausentes para variáveis numéricas! No entanto, ainda precisamos substituir os valores ausentes em variáveis do tipo categóricas — o que veremos ainda neste capítulo.

Substituição de valores ausentes pela mediana utilizando a biblioteca feature-engine

Da mesma forma que na biblioteca *scikit-learn*, uma das facilidades que a *feature-engine* traz é a de que, para fazermos a substituição de valores ausentes pela mediana, o código será muito similar ao da média, mudando apenas o parâmetro `imputation_method` de `mean` para `median`. Veja o seguinte código:

```
from feature_engine.imputation import MeanMedianImputer
imputer_mediana = MeanMedianImputer(imputation_method="median",variables=
["PassengerId","Pclass","Age","SibSp","Fare"])
```

Perceba nesse código que as variáveis que transformaremos são as mesmas do método passado. Sendo assim, para você conseguir substituir os valores ausentes pela mediana, é preciso que você tenha valores ausentes em seu conjunto de treino e conjunto de teste.

Vamos treinar o nosso `MeanMedianImputer` passando o conjunto de `numericos_treino` com valores faltantes e fazer as transformações

necessárias, isto é, substituir os valores ausentes pela mediana:

```
imputer_mediana.fit(numericos_treino)
numericos_treino = imputer_mediana.transform(numericos_treino)
numericos_teste = imputer_mediana.transform(numericos_teste)
```

Vamos aproveitar para inspecionar os valores aprendidos pelo nosso `MeanMedianImputer`, com o comando `imputer_mediana.imputer_dict_`. Ao executarmos esse comando, temos como retorno o seguinte dicionário:

```
{'PassengerId': 447.6789727126806,
 'Pclass': 2.287319422150883,
 'Age': 29.915338645418327,
 'SibSp': 0.5313001605136437,
 'Fare': 32.45827255216693}
```

Como na média, aqui o retorno do `imputer_mediana.imputer_dict_` é um dicionário de dados. Para terminar, podemos usar o comando `numericos_treino.isnull().sum()` para conferir se temos dados ausentes no nosso conjunto de treino nas colunas que modificamos:

```
PassengerId    0
Pclass         0
Age            0
SibSp          0
Fare           0
dtype: int64
```

E para conferir se há valores ausentes no nosso conjunto de teste, usamos o comando `numericos_teste.isnull().sum()`:

```
PassengerId    0
Pclass         0
Age            0
SibSp          0
Parch          0
Fare           0
dtype: int64
```

Com isso, conseguimos novamente substituir os valores ausentes do tipo numérico. Na próxima parte, veremos como fazer a substituição de valores ausentes do tipo categórico.

Substituindo valores ausentes de um conjunto de dados categóricos utilizando a biblioteca scikit-learn

Nesta seção, veremos como utilizar a biblioteca *scikit-learn* para realizar a substituição de valores ausentes em um conjunto de dados categóricos. Vamos utilizar os nossos conjuntos de treino e teste que temos utilizado até agora. Lembre-se: seu dataframe deve ter valores ausentes nas categorias com que vamos trabalhar.

Para fazer a imputação de valores ausentes em dados categóricos, vamos recordar quais são as colunas de dados categóricos que serão trabalhadas. Executando o comando `X_treino.dtypes`, temos como retorno os tipos de cada uma das colunas do nosso dataframe:

```
PassengerId  int64
Pclass       int64
Name         object
Sex          object
Age          float64
SibSp        int64
Parch        float64
Ticket       object
Fare         float64
Cabin        object
Embarked     object
dtype: object
```

Então, os nossos dados categóricos são: Name, Sex, Ticket, Cabin e Embarked.

Lembrando que, dessas, as colunas com dados ausentes são as colunas `Sex`, com `0.051270`, e `Cabin`, com `0.791096`. Porém, antes de começarmos a fazer as substituições necessárias nas colunas faltantes, repare que a coluna `Cabin` possui 79% de dados ausentes.

Não há muito o que fazer neste caso: devido à quantidade de dados ausentes, vamos ter que excluir essa coluna e não a utilizar em um futuro modelo de *machine learning* que formos criar com esses dados.

Para excluir a coluna `Cabin`, podemos utilizar o seguinte código:

```
X_treino.drop(columns=["Cabin"],inplace=True)
X_teste.drop(columns=["Cabin"],inplace=True)
```

> **Exclusão de dados**: podemos utilizar o `.drop` para excluirmos as colunas que possuem muitos valores ausentes. Embora não tenhamos uma definição sobre o que é uma grande quantidade de valores ausentes, por convenção utilizamos a exclusão de colunas quando o valor de dados ausentes for superior a 75%.

Desta forma, não só apagamos a coluna `Cabin` do nosso conjunto `X_treino` e `X_teste` como tornamos essa mudança permanente com o parâmetro `inplace=True`.

Agora vamos fazer a substituição dos valores ausentes na coluna `Sex`, mas, antes, vamos averiguar quantas categorias temos em nossa coluna. Para isso, vamos utilizar o comando da biblioteca `unique` na coluna `Sex`. Para os dados de treino, podemos fazer `X_treino["Sex"].unique()`, e o resultado da execução desse código pode ser visto a seguir:

```
array(['male', 'female', nan], dtype=object)
```

Faremos a mesma coisa para o dataframe de teste, `X_teste["Sex"].unique()`:

```
array([nan, 'male', 'female'], dtype=object)
```

Conseguimos perceber que, fora a categoria `NaN`, que descreve o valor ausente, temos apenas duas categorias, `female` e `male`, em ambos os conjuntos.

Agora vamos utilizar o *scikit-learn* para fazer as substituições necessárias. O método utilizado para fazer essa substituição é o mesmo que utilizamos para a média e para a mediana: o método `SimpleImputer`. Da mesma forma, devemos separar os valores com os quais trabalharemos, ou seja, os valores categóricos, pois se passarmos valores numéricos receberemos um erro.

Para separarmos a coluna `Sex`, podemos utilizar o código a seguir:

```
sex_treino= X_treino["Sex"]
sex_teste = X_teste["Sex"]
```

Podemos verificar os valores categóricos salvos em dois dataframes distintos. Primeiro, vamos verificar o nosso conjunto `sex_treino` utilizando o comando `sex_treino.head()`:

```
857      male
52     female
386      male
124      male
578       NaN
Name: Sex, dtype: object
```

Vamos verificar o nosso conjunto `sex_teste` com o comando `sex_teste.head()`:

```
495       NaN
648      male
278      male
31        NaN
255    female
Name: Sex, dtype: object
```

Agora podemos fazer as mudanças necessárias. Para isso, vamos importar o método `SimpleImputer` da biblioteca `sklearn.impute` e, em seguida, instanciá-lo passando o parâmetro `strategy=most_frequent`, para que o método instanciado aprenda a categoria frequente (moda) do conjunto de dados passado para ele. Veja o código a seguir:

```
from sklearn.impute import SimpleImputer
imputer_freq = SimpleImputer(strategy="most_frequent")
```

> Quando utilizamos como estratégia de substituição o parâmetro `most_frequent`, estamos mostrando ao `SimpleImputer` que desejamos fazer a substituição dos valores ausentes pela moda.

Com o `SimpleImputer` instanciado, vamos treiná-lo utilizando o conjunto de dados de treino `sex_treino`, como no código a seguir:

```
imputer_freq.fit(sex_treino.to_frame())
```

Precisamos passar o comando `to_frame()` dentro do método `fit` porque esse método da biblioteca *scikit-learn* recebe apenas dataframes; como o nosso conjunto de dados possui apenas uma coluna, o método `.fit` o enxerga como uma *series*. Então, ao passar `to_frame()`, a *series* `sex_treino` passa a ser um dataframe e não recebemos erro nenhum.

Da mesma forma como fizemos com os valores numéricos, podemos utilizar o comando `imputer_freq.statistics_` para inspecionarmos o que foi aprendido pelo nosso `SimpleImputer`. Ao executarmos o comando `imputer_freq.statistics_`, temos como resultado:

```
array(['male'], dtype=object)
```

Com isso, vemos que, em nosso conjunto, a categoria que mais se repete (ou seja, a *moda*) é o gênero `male`. Agora podemos transformar nossos dados com o comando `transform`, da seguinte forma:

```
sex_treino = imputer_freq.transform(sex_treino.to_frame())
sex_teste = imputer_freq.transform(sex_teste.to_frame())
```

O *scikit-learn* transforma o nosso conjunto de dados em um `numpy array`, então vamos transformá-lo em um dataframe novamente:

```
sex_treino =pd.DataFrame(sex_treino,columns=["Sex"])
sex_teste =pd.DataFrame(sex_teste,columns=["Sex"])
```

Com isso, conseguimos utilizar o comando `isnull().sum()` para contabilizar os valores ausentes em `sex_treino` e em `sex_teste`. Ao executarmos o comando `sex_treino.isnull().sum()`, temos o seguinte retorno:

```
Sex    0
dtype: int64
```

Ou seja, não temos valores ausentes. Obtemos o mesmo resultado para o nosso conjunto de teste ao executarmos o comando `sex_teste.isnull().sum()`:

```
Sex    0
dtype: int64
```

Substituindo os valores ausentes de um conjunto de dados categóricos utilizando a biblioteca feature-engine

Diferente da biblioteca *scikit-learn*, a biblioteca *feature-engine* não utiliza o `MeanMedianImputer` para fazer a substituição dos valores categóricos, mas, sim, o comando `CategoricalImputer`. Vamos então importar essa classe:

```
from feature_engine.imputation import CategoricalImputer
```

Podemos agora instanciar nossa classe `CategoricalImputer`, mas em seu `imputation_method` vamos inserir a palavra `frequent` para que o `CategoricalImputer` aprenda a categoria frequente. Passaremos também a coluna que queremos transformar no parâmetro `variables`. Veja como ficará o código:

```
ft_imputer = CategoricalImputer(imputation_method="frequent",variables=
["Sex"])
```

Agora vamos treinar o nosso `CategoricalImputer` com o comando `.fit`:

```
ft_imputer.fit(X_treino)
```

Vamos inspecionar o que foi aprendido pelo `CategoricalImputer`. Ao executarmos o comando `ft_imputer.imputer_dict_`, temos o seguinte retorno:

```
{'Sex':'male'}
```

O retorno foi um dicionário com um único valor, correspondente à única coluna com que estamos trabalhando. Agora podemos fazer a transformação dos nossos dados:

```
X_treino = ft_imputer.transform(X_treino)
X_teste= ft_imputer.transform(X_teste)
```

Como a biblioteca *feature-engine* já devolve para nós um dataframe, não há necessidade de fazer nenhuma conversão. Por fim, podemos fazer a contagem dos valores ausentes da coluna `Sex`, que foi a coluna que trabalhamos nessa parte. Para isso, basta executar os seguintes comandos:

- Para os dados de treino, `X_treino["Sex"].isnull().sum()`. O resultado da execução desse código é `0`;
- Para os dados de teste, `X_teste["Sex"].isnull().sum()`. Ao executarmos esse código, temos como retorno `0`.

Pronto! Agora não há mais nenhum valor ausente em nosso dataframe! Com isso, encerramos esta primeira parte de nosso projeto prático, porém há muito trabalho a se fazer aqui: nosso conjunto de dados do Titanic ainda não está pronto para ser usado em algoritmos de *machine learning*. Veremos nos próximos capítulos outros ajustes que podem ser feitos nessa base de dados.

PARTE 2: Trabalhando com valores numéricos

CAPÍTULO 3
Trabalhando com valores numéricos

> O notebook e os materiais para este capítulo e para o projeto prático podem ser encontrados em: materiais da parte 2 (https://github.com/ThiagoBenevides/feature-engine-transformando-dados-brutos-em-recursos/tree/main/parte_dois)

Valores numéricos, sejam eles *discretos* ou *contínuos*, são os mais importantes e comuns quando falamos em Ciência de Dados. Esses valores podem vir de fontes inesgotáveis de locais, por exemplo, sensores de temperatura, balanças, câmeras de segurança, entre outros.

Embora valores numéricos já estejam em um formato de fácil consumo por algoritmos de *machine learning*, isso não significa que não há trabalho a ser feito com eles.

Quando falamos em recursos, mais especificamente em *características de recursos*, podemos pensar que características não devem apenas representar aspectos visíveis dos nossos dados, mas também devem estar de acordo com as premissas do modelo que vamos utilizar.

Por exemplo, caso façamos uso de algoritmos que utilizam modelos lineares para fazerem suas previsões, devemos ter em mente que os dados que serão consumidos por esse modelo devem estar de acordo com as premissas de linearidade. Dessa forma, as transformações feitas em conjuntos numéricos em boa parte dos

casos são necessárias para tornarmos os dados mais fáceis de serem consumidos pelos algoritmos de *machine learning*.

Ao trabalharmos com valores numéricos, podemos pensar em algumas perguntas em relação ao nosso conjunto de dados:

1. Qual é a **magnitude** dos dados, isto é, quão pequenos ou quão grandes eles são? Essa pergunta deve ser feita pois alguns algoritmos de *machine learning* são extremamente sensíveis à magnitude dos números, por exemplo, algoritmos que utilizam modelos lineares.
2. Qual é o **domínio** dos nossos dados? Ter essa informação pode ser útil para ajudar a localizar erros de inserção em banco de dados. Por exemplo, caso nosso conjunto de valores representem as idades de consumidores, sabemos que, se houver valores negativos, trata-se de um erro.

Agora que compreendemos o motivo de trabalharmos com valores numéricos, vamos ao primeiro processo, chamado de *binning*.

3.1 Binning, a discretização de valores numéricos

O processo de *binning*, também chamado de *binarização* ou *discretização*, consiste em criar "caixas" (ou *bins*) e distribuir os valores do conjunto de dados entre as caixas criadas.

Para facilitar o raciocínio do processo de discretização, vamos considerar a seguinte tabela para fazermos um exemplo:

Nome	Idade	Saldo
Miguel	75	9734.05
Arthur	42	12517.09
Heitor	50	6304.24
Helena	61	11118.21
Alice	77	9542.27
Theo	35	2964.03
Davi	84	13232.54
Lilia	31	11684.21
Laura	84	4919.49
Gabriel	64	5293.05
Bernardo	54	8847.02
Samuel	67	1789.03
Valentina	69	11113.87
Heloísa	22	11665.19
Enzo	63	2432.18
Pedro	82	11626.03
Lorenzo	51	10038.05
Sophia	63	1690.01
Maria	56	6409.31
Eduarda	87	13546.88
Cecília	32	1298.90
Júlia	65	1200.00

Nome	Idade	Saldo
Isabela	54	16000.00
Benício	53	5000.00

A tabela que acabamos de ver apresenta 24 pessoas diferentes com o nome, a idade e o saldo em conta corrente. Observando a coluna `Saldo` da tabela anterior, você verá que o saldo é um valor contínuo, ou seja, pode assumir valores reais.

Esses valores podem prejudicar o nosso algoritmo de *machine learning* pelo fato de serem vários números a serem considerados pelo nosso modelo. Podemos utilizar o processo de discretização para reduzir a quantidade de valores na coluna `Saldo`.

Como exemplo, criaremos quatro intervalos (ou quatro "caixas") para discretizarmos a coluna `Saldo`, da seguinte forma:

- Primeiro intervalo: (1200.00,4900.00]
- Segundo intervalo: (4900.00,8600.00]
- Terceiro intervalo: (8600.00,12300.00]
- Quarto intervalo: (12300.00,16000.00]

Observação: você deve ter reparado que os intervalos estão com um sinal de parêntese à esquerda e colchete à direita. Essa simbologia na matemática serve para mostrar que à direita temos um intervalo aberto e, à esquerda, um intervalo fechado. Isso acontece para não termos um problema de um mesmo número pertencer a dois intervalos distintos ao mesmo tempo.

Nome	Idade	Saldo
Miguel	75	3
Arthur	42	3

Nome	Idade	Saldo
Heitor	50	2
Helena	61	2
Alice	77	3
Theo	35	1
Davi	84	4
Lilia	31	3
Laura	84	2
Gabriel	64	2
Bernardo	54	2
Samuel	67	1
Valentina	69	3
Heloísa	22	3
Enzo	63	1
Pedro	82	3
Lorenzo	51	3
Sophia	63	1
Maria	56	2
Eduarda	87	4
Cecília	32	1
Júlia	65	1
Isabela	54	4
Benício	53	2

Observe que, dos 24 valores diferentes para o saldo, agora temos apenas 4 valores diferentes, tornando o nosso conjunto de dados mais simples e leve para ser consumido por modelos de *machine learning*.

> **OBSERVAÇÃO:** a forma de discretização que acabamos de ver aqui é apenas uma de muitas formas. Existem algumas outras que permitem que façamos a melhor escolha possível para o número de compartimentos. Aqui, o valor de 4 compartimentos foi escolhido de forma aleatória.

Discretizando valores por largura de intervalo igual

A primeira técnica de discretização que veremos é a mais simples de todas e foi utilizada no nosso exemplo anterior. Chama-se *discretização por largura de intervalo igual* e consiste em calcularmos uma largura de intervalo que será constante para todos os intervalos criados.

A fórmula para fazer o cálculo da largura em Python é a seguinte:

```
import numpy as np

largura = (np.min(x) - np.max(x))/bins
```

O código que acabamos de escrever faz o seguinte:

- `np.min(x)` : usa a função `np.min()` da biblioteca *numpy* para encontrar o valor mínimo no array `x` . O array `x` é o conjunto de dados para o qual você deseja criar um histograma;
- `np.max(x)` : da mesma forma, usamos a função `np.max()` para encontrar o valor máximo no array `x` ;
- `bins` : esse é um valor que representa o número de intervalos (ou *bins*) em que você deseja dividir seus dados no histograma.

Isso controla a quantidade de detalhe ou granularidade do histograma.

Para operarmos a fórmula em nosso exemplo, precisamos selecionar os valores mínimo e máximo da coluna `saldo`, que são, respectivamente, `1200.00` e `16000.00`. Para o valor de `N`, que indicará o número de *bins*, usaremos o número `4` para ficar semelhante ao exemplo dado no início da seção.

Com o valor da largura calculado, podemos dar início ao cálculo dos intervalos. Para isso, vamos prosseguir da seguinte forma: para o primeiro intervalo, selecionamos o menor valor da nossa tabela que, neste caso, é `1200.00`, e somamos o valor da largura, que é de `3700.00`, resultando no valor `4900.00`. Dessa forma, nosso primeiro intervalo possui como limites inferior e superior os valores `1200.00` e `4900.00`, respectivamente. Para o segundo intervalo, vamos selecionar o limite superior do primeiro intervalo e adicionamos mais uma vez o valor da largura. Isso resultará no valor de `8600.00`, e assim prosseguimos calculando os demais intervalos.

- Intervalo 1: (1200.00,4900.00]
- Intervalo 2: (4900.00,8600.00]
- Intervalo 3: (8600.00,12300.00]
- Intervalo 4: (12300.00,16000.00]

Nome	Idade	Saldo
Miguel	75	3
Arthur	42	3
Heitor	50	2
Helena	61	2
Alice	77	3
Theo	35	1

Nome	Idade	Saldo
Davi	84	4
Lilia	31	3
Laura	84	2
Gabriel	64	2
Bernardo	54	2
Samuel	67	1
Valentina	69	3
Heloísa	22	3
Enzo	63	1
Pedro	82	3
Lorenzo	51	3
Sophia	63	1
Maria	56	2
Eduarda	87	4
Cecília	32	1
Júlia	65	1
Isabela	54	4
Benício	53	2

Agora que conseguimos compreender melhor como esse processo de discretização funciona, pense, por exemplo, em como uma pessoa que trabalha em um banco faz a discretização durante o seu trabalho. Obviamente, ela não imprime uma planilha com todos os clientes e faz todas as contas na mão: isso seria impraticável.

Levaria muitas horas ou até dias, dependendo do número de clientes deste banco. O que faremos agora é *automatizar* esse processo com código.

Vamos começar importando a biblioteca *pandas* para criarmos a nossa tabela em forma de dataframe. Em seguida, criaremos três listas contendo as seguintes informações: `nome`, `idade` e `saldo` dos clientes. Por fim, uniremos essas três listas em um dataframe:

```
import pandas as pd

nome =
["Miguel","Arthur","Heitor","Helena","Alice","Theo","Davi","Lilia","Laura"
,"Gabriel","Bernardo","Samuel","Valentina","Heloísa","Enzo","Pedro","Loren
zo","Sophia","Maria","Eduarda","Cecília","Júlia","Isabella","Benício"]
idade =
[75,42,50,61,77,35,84,31,84,64,54,67,69,22,63,82,51,63,56,87,32,65,54,53]
saldo =
[9734.05,12517.09,6304.24,11118.21,9542.27,2964.03,13232.54,11684.21,4919.
49,5293.05,8847.02,1789.03,11113.87,11665.19,2432.18,11626.03,10038.05,169
0.01,6409.31,13546.88,1298.90,1200.00,16000.00,5000.00]
dataframe = pd.DataFrame({"Nome":nome,"Idade":idade,"Saldo":saldo})
```

O resultado dessa codificação pode ser visualizado com o comando `dataframe.head(10)`, que tem como retorno a seguinte tabela com os 10 primeiros valores:

Nome	Idade	Saldo
Miguel	75	9734.05
Arthur	42	12517.09
Heitor	50	6304.24
Helena	61	11118.21
Alice	77	9542.27
Theo	35	2964.03

Nome	Idade	Saldo
Davi	84	13232.54
Lilia	31	11684.21
Laura	84	4919.49
Gabriel	64	5293.05

Vamos discretizar a nossa coluna `Saldo`. Primeiro, faremos todos os cálculos via Python e, no final, criaremos uma função que discretiza toda a coluna.

O primeiro passo é capturar os valores máximo e mínimo do nosso dataframe; para isso, utilizaremos os métodos `max()` e `min()` da biblioteca *pandas*, como no código a seguir:

```
minimo = dataframe["Saldo"].min()
maximo = dataframe["Saldo"].max()
```

As variáveis que acabamos de criar, `minimo` e `maximo`, armazenam os números `1200.00` e `16000.00`, respectivamente. Agora vamos calcular a largura do nosso intervalo, da seguinte forma:

```
bins = 4
largura = (maximo-minimo)/bins
```

Os cálculos foram feitos da mesma forma como no exemplo do início do capítulo, e o resultado da largura é o número `3700.00`.

Para calcular os intervalos, utilizaremos uma *list comprehension* para fazermos uma lista que servirá de base para a criação dos nossos intervalos:

```
intervalos = [i for i in range(minimo,maximo+largura,largura)]
```

Porém, ao executarmos essa linha de comando, teremos um erro dizendo que `list comprehension` só aceita números inteiros. Esse erro acontece, pois o *range* utilizado dentro da *list comprehension*

considera apenas valores inteiros. Então, para evitarmos esse erro, devemos fazer o seguinte ajuste:

- para o `minimo`, faremos a inserção do método `np.floor` no código para arredondarmos para baixo e, em seguida, converteremos o resultado em um número inteiro;
- para o `maximo`, vamos inserir o método `np.ceil` para arredondarmos o valor para cima e, em seguida, o converteremos para um número do tipo inteiro.

```
minimo = int(np.floor(dataframe["saldo"].min()))
maximo = int(np.ceil(dataframe["saldo"].max()))
largura = int((maximo-minimo)/4)
```

Executando o nosso comando para criar os intervalos, teremos os seguinte retorno: `[1200,4900,8600,12300,16000]`.

Por fim, vamos utilizar o método `cut` da biblioteca *pandas* para discretizarmos a nossa coluna `saldo`. No método `cut`, inserimos primeiro o dataframe com os valores que queremos manipular — em nosso caso, é o `dataframe` que criamos com a coluna `saldo`. Em seguida, passamos como `bins` a lista com os intervalos calculados. Isso criará nossa discretização. Por fim, passamos `include_lowest` como `True` para que ele inclua o primeiro intervalo em nosso conjunto.

```
dataframe["saldo_discretizado"] =
pd.cut(dataframe["saldo"],bins=intervalos,include_lowest=True)
```

Dessa forma, conseguimos discretizar a nossa coluna `saldo` em uma outra chamada `saldo_discretizado`. O resultado dessa discretização pode ser visto com o comando `dataframe.head()`, que retorna a seguinte tabela:

index	Nome	Idade	Saldo	saldo_discretizado
0	Miguel	75	9734.05	(9194.645, 11635.82]

index	Nome	Idade	Saldo	saldo_discretizado
1	Arthur	42	12517.09	(11635.82, 16000.0]
2	Heitor	50	6304.24	(4430.625, 9194.645]
3	Helena	61	11118.21	(9194.645, 11635.82]
4	Alice	77	9542.27	(9194.645, 11635.82]

Agora vamos criar uma função que concentra esse mesmo código; assim, quando quisermos discretizar um dataframe, basta chamarmos a função passando para ela o dataframe, a coluna e o número de intervalos que queremos criar:

```
def disc_intervalo(dataframe,variavel,bins):
    maximo = int(np.floor(dataframe[variavel].max()))
    minimo = int(np.ceil(dataframe[variavel].min()))
    largura = int((maximo-minimo)/bins)

    intervalos = [i for i in range(minimo,maximo+largura,largura)]
    intervalos = intervalos[0:len(intervalos)-1]
    intervalo_discretizado = pd.cut(dataframe[variavel],bins=intervalos,
include_lowest=True)

    return intervalo_discretizado
```

Agora que já construímos a discretização dos dados e criamos uma função para automatizar esse trabalho, vamos analisar os resultados obtidos com o processo. Vamos criar um gráfico que faz a contagem de observações em cada um dos intervalos. Com ele, conseguiremos ver a proporção de cada intervalo e diagnosticar possíveis problemas na discretização. Para criá-lo, vamos utilizar o seguinte código:

```
import matplotlib.pyplot as plt
%matplotlib inline

dataframe.groupby("saldo_discretizado")["saldo"].count().plot.bar()
```

```
plt.xticks(rotation=45)
plt.ylabel('Número de observações em cada intervalo')
```

A BIBLIOTECA MATPLOTLIB

A biblioteca *matplotlib* pode ser utilizada para criarmos uma ampla variedade de gráficos, incluindo gráficos de linhas, de barras, histogramas, gráficos de dispersão, gráficos de pizza, entre outros. O *matplotlib* é uma das escolhas mais populares para visualização de dados em Python, sendo usado por uma ampla gama de usuários, e isso se deve ao fato de que ele é simples de ser utilizado, possui uma boa documentação e também possui uma comunidade.

A documentação do *matplotlib* está disponível em: https://matplotlib.org/stable/index.html.

O resultado pode ser visto na figura a seguir:

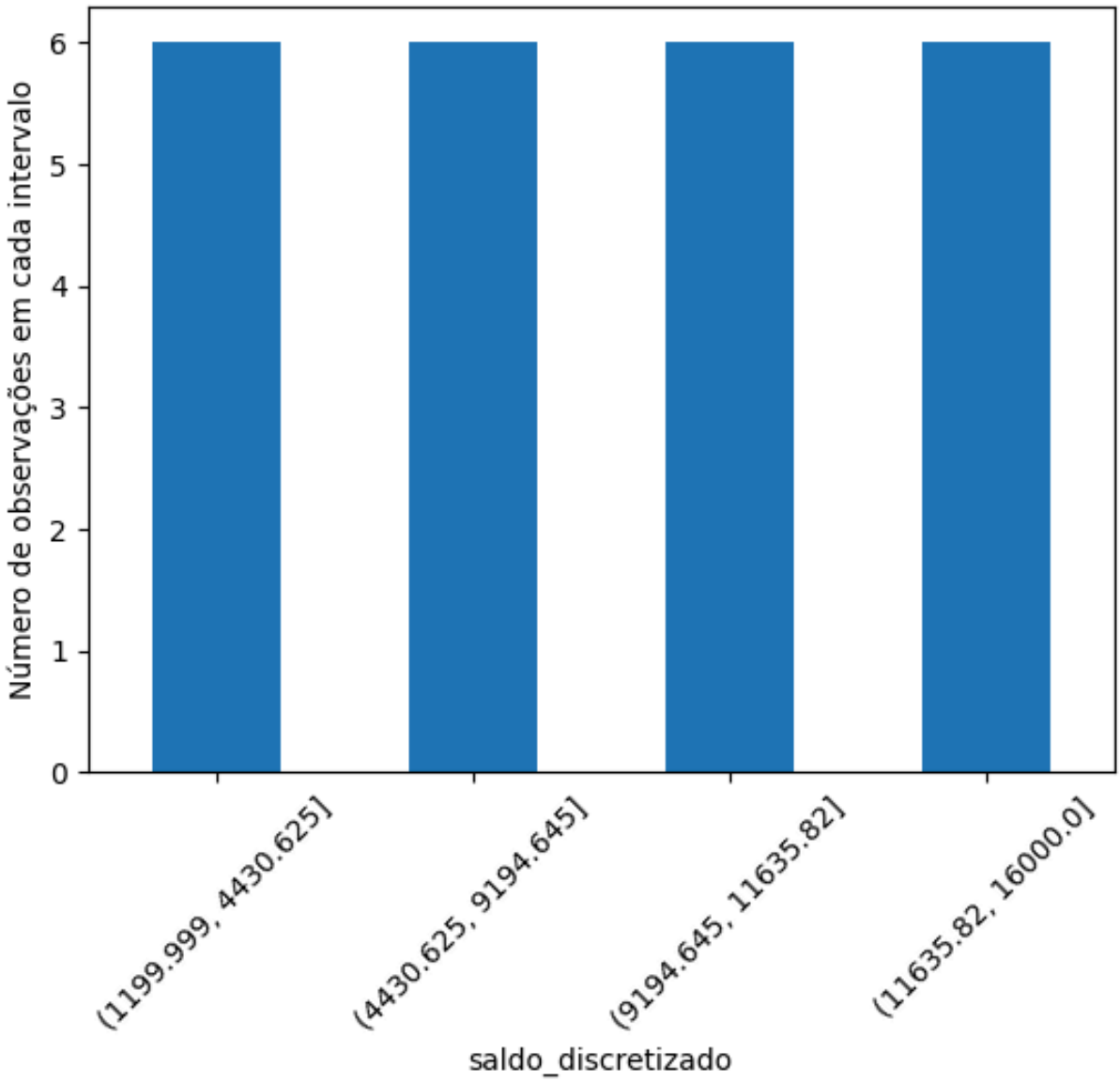

Figura 3.1: Resultado do gráfico que exibe a proporção de observações em cada um dos intervalos criados.

Observe que, neste conjunto de dados reduzido que utilizamos, nossa técnica de discretização conseguiu equilibrar os dados de maneira perfeita. Isso significa que todos os intervalos (*bins*) ficaram com o mesmo número de elementos, entretanto, é importante ressaltar que isso nem sempre acontece, como veremos mais adiante em nosso projeto prático.

Esse processo de discretização é muito sensível a outliers. Caso seu conjunto de dados possua muitos valores assim, você terá que tratá-los antes de discretizar seu conjunto de dados utilizando a largura igual. Falaremos mais à frente sobre *outliers* e como tratá-los.

Discretizando valores numéricos com intervalo de frequência igual

Como vimos, a discretização de largura igual pode provocar certas distorções, como a distribuição no número de observações em cada intervalo. Para conseguirmos contornar esse problema, vamos ver uma outra forma de discretização.

O método que veremos utiliza a ideia de *percentil*, *decil* e *quartil* para criar os intervalos com números de observações proporcionais. De forma simples, esse método ordena os valores em ordem decrescente e divide da seguinte maneira:

- Percentil: são criados 100 intervalos proporcionais;
- Decil: são utilizados 10 intervalos proporcionais;
- Quartil: são utilizados 4 intervalos proporcionais.

Devido a esses conceitos fugirem do escopo deste livro, não faremos aqui o cálculo manual da discretização; partiremos para o código em Python — assim ficará mais fácil de entender os conceitos apresentados.

Vamos começar importando as bibliotecas e criando o dataframe que utilizaremos nesse processo de discretização:

```
import pandas as pd

nome =
["Miguel","Arthur","Heitor","Helena","Alice","Theo","Davi","Lilia","Laura"
,"Gabriel","Bernardo","Samuel","Valentina","Heloísa","Enzo","Pedro","Loren
zo","Sophia","Maria","Eduarda","Cecília","Júlia","Isabella","Benício"]
```

```
idade =
[75,42,50,61,77,35,84,31,84,64,54,67,69,22,63,82,51,63,56,87,32,65,54,53]
saldo =
[9734.05,12517.09,6304.24,11118.21,9542.27,2964.03,13232.54,11684.21,4919.
49,5293.05,8847.02,1789.03,11113.87,11665.19,2432.18,11626.03,10038.05,169
0.01,6409.31,13546.88,1298.90,1200.00,16000.00,5000.00]

dataframe = pd.DataFrame({"Nome":nome,"Idade":idade,"Saldo":saldo})
```

Para a discretização com a faixa de frequência, utilizaremos um método *pandas* semelhante ao utilizado no exemplo no qual tratamos de faixa de valor igual. Porém, em vez de `cut`, utilizaremos o `qcut`, que é uma função da biblioteca *pandas* usada para agrupar dados numéricos em quantis de tamanho igual.

O QUE É O PD.QCUT?

O `pd.qcut` é usado para dividir uma coluna de dados numéricos em intervalos ou categorias com base nos quantis. Ele é particularmente útil quando se deseja criar categorias de tamanho uniforme com base na distribuição dos dados. Ao utilizarmos o `pd.qcut`, podemos especificar o número de quantis ou o número de intervalos desejados. A função calcula os quantis para os seus dados e os usa para dividir a coluna em intervalos aproximadamente iguais. Isso significa que cada intervalo conterá aproximadamente o mesmo número de observações. O `pd.qcut` é uma abordagem útil quando desejamos criar grupos de tamanhos semelhantes, independente dos valores específicos dos dados.

```
dataframe["saldo_discretizado"], intervalos =
pd.qcut(dataframe["saldo"],4, labels=None,retbins=True)
```

Vamos entender o que são esses valores passados no `qcut`. O primeiro é o dataframe com a coluna, nesse caso é o dataframe que criamos na parte anterior do código com a coluna `saldo`. O segundo parâmetro é o número de intervalos que queremos criar, que, para

esse exemplo, serão `4`. O parâmetro `labels` é utilizado para que sejam retornados os rótulos. Para esse caso, não haverá retorno, pois passamos como `None` como parâmetro. Por fim, passamos o `retbins=True` para que retorne os intervalos com números inteiros.

Você deve ter percebido que, diferente do método `cut`, o método `qcut` retornou dois valores. O primeiro deles é o intervalo discretizado, como mostra a tabela a seguir, resultado do comando `.head()`:

Nome	Idade	Saldo	saldo_discretizado
Miguel	75	9734.05	(9194.645, 11635.82]
Arthur	42	12517.09	(11635.82, 16000.0]
Heitor	50	6304.24	(4430.625, 9194.645]
Helena	61	11118.21	(9194.645, 11635.82]
Alice	77	9542.27	(9194.645, 11635.82]
Theo	35	2964.03	(1199.999, 4430.625]
Davi	84	13232.54	(11635.82, 16000.0]
Lilia	31	11684.21	(11635.82, 16000.0]
Laura	84	4919.49	(4430.625, 9194.645]
Gabriel	64	5293.05	(4430.625, 9194.645]

O outro, chamado de `saldo_discretizado`, são os limites de cada intervalo criado, semelhantes aos intervalos criados manualmente pela faixa de largura igual:

- (1199.999, 4430.625]
- (4430.625, 9194.645]
- (9194.645, 11635.82]
- (11635.82, 16000.0]

Para finalizar, vamos construir o gráfico com a proporção de cada uma das observações. Pelo fato de termos construído 4 intervalos, conseguiremos comparar com o método anterior.

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(figsize=(12,7))
dataframe.groupby("saldo_discretizado")["idade"].count().plot.bar()
plt.xticks(rotation=45)
plt.ylabel('Número de observações em cada intervalo pelo método de frequência igual')
```

O resultado do código pode ser visto na figura a seguir:

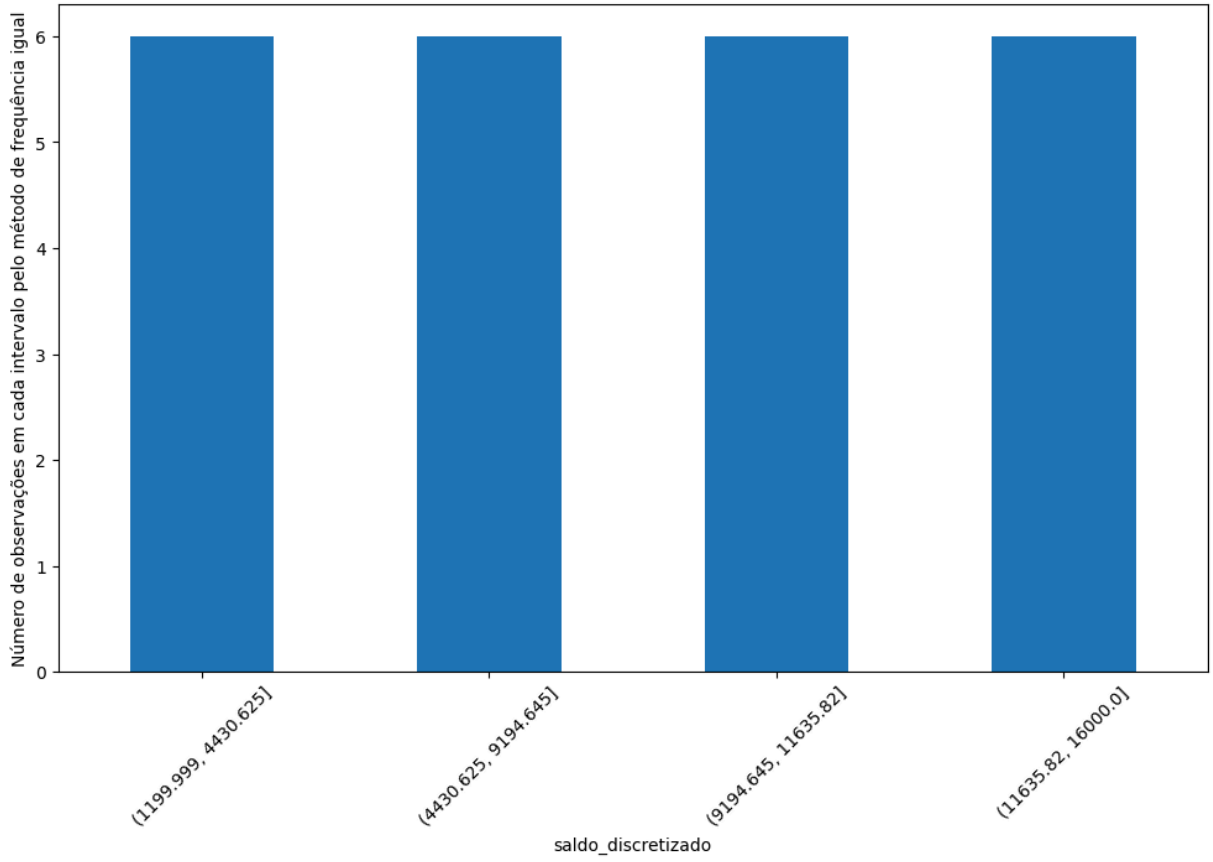

Figura 3.2: Número de observações em cada intervalo pelo método de frequência igual.

Perceba que, neste caso em particular, temos o mesmo número de observações em cada intervalo, mas isso nem sempre ocorre. Devido à distribuição dos nossos dados, podemos ter variações entre as observações, embora o método utilizado fique mais proporcional ao da discretização por faixa de valor igual, como veremos no projeto prático do próximo capítulo.

Um outro fator importante ao qual devemos ter atenção ao utilizarmos o `pd.qcut` é em relação à distribuição dos nossos dados: caso existam lacunas em nossa distribuição (espaços ou intervalos na distribuição dos valores), podemos criar intervalos vazios, ou seja, sem nenhum valor observado.

Vamos concluir a discretização por faixa de frequência criando uma função para automatizar nossa discretização. A função receberá três parâmetros, sendo eles: o dataframe que vamos manusear, a coluna a ser transformada e o número de *bins* ou intervalos que queremos criar:

```
def disc_freq(dataframe, variavel,q):
        maximo = dataframe[variavel].max()
        minimo = dataframe[variavel].min()

        largura = int((maximo-minimo)/bins)
        intervalos = [i for i in range(minimo,maximo+largura,largura)]
        discretizado,intervaldisc =
pd.qcut(dataframe[variavel],q,labels=None,retbins=True)

        return discretizado, intervaldisc
```

O QUE É O PARÂMETRO `q`, UTILIZADO NESSE ÚLTIMO CÓDIGO?

O parâmetro `q` especifica o número de quantis a serem usados. O valor padrão para ele é `q=10`, ou seja, se não especificarmos nenhum valor para `q`, ele usará o valor 10 de forma automática, o que significa que os dados serão divididos em 10 quantis de tamanho igual. Então, podemos observar que o parâmetro `q` especificará o número de conjuntos de tamanhos iguais que serão criados.

Observe que, da mesma forma como o método `qcut` retorna dois valores, a nossa função devolverá dois valores, sendo um o dataframe transformado e o outro, a lista com os limites calculados.

Discretizando um conjunto de dados numéricos utilizando k-means

As duas técnicas que discutimos até agora têm como objetivo reduzir o espaço de recursos, viabilizando o uso de algoritmos de aprendizado de máquina que são baseados em categorias, mesmo no caso de os valores serem numéricos contínuos. Entretanto, devemos ressaltar que ambas as técnicas que vimos são classificadas como métodos supervisionados. Isso significa que esses métodos necessitam da orientação do chamado supervisor, que é a pessoa responsável por determinar o número de intervalos desejados e como esses intervalos serão construídos.

Já as duas técnicas que veremos agora são conhecidas como **métodos não supervisionados**. Um método não supervisionado não precisa da figura do supervisor para definir o número de intervalos que serão criados, pois quem decidirá o tamanho do intervalo é o próprio algoritmo.

A primeira técnica que abordaremos utiliza o algoritmo conhecido como *k-means*. O *k-means* baseia-se no conceito de instância sintética. Em vez de calcularmos as distâncias entre todas as

instâncias do cluster *k-means*, calculamos apenas as distâncias aos seus respectivos centroides. Grosso modo, o algoritmo de *k-means* utiliza duas etapas para a criação dos clusters:

1. **Etapa de inicialização**: nesta etapa, *k* observações são escolhidas de forma aleatória como centros iniciais dos *k* clusters que são identificados pelo algoritmo de *k-means*. Embora existam técnicas mais aprimoradas para fazer a escolha inicial para o valor de *k*, neste primeiro momento vamos considerar a aleatoriedade como escolha de *k*;
2. **Etapa de iteração**: os centros dos clusters são recalculados com os pontos médios de todas as observações dentro do cluster mais próximo recém-criado.

Essas duas etapas são feitas de forma recursiva até que sejam encontrados os chamados centros ótimos dos clusters. Assim como o método anterior, aqui também não entraremos nos cálculos utilizados para os clusters. Em vez disso, partiremos direto para o código. Entretanto, a biblioteca *pandas* não possui método para fazer esse tipo de discretização; sendo assim, para efetuar o *k-means*, usaremos a biblioteca *scikit-learn*.

Para compreendermos melhor como o algoritmo de *k-means* funciona, vamos utilizá-lo em uma base de dados juntamente da biblioteca *scikit-learn*. Para isso, vamos importar as bibliotecas que serão utilizadas e criar o dataframe que será utilizado para o nosso exemplo:

```
import pandas as pd

nome =
["Miguel","Arthur","Heitor","Helena","Alice","Theo","Davi","Lilia","Laura"
,"Gabriel","Bernardo","Samuel","Valentina","Heloísa","Enzo","Pedro","Loren
zo","Sophia","Maria","Eduarda","Cecília","Júlia","Isabella","Benício"]
idade =
[75,42,50,61,77,35,84,31,84,64,54,67,69,22,63,82,51,63,56,87,32,65,54,53]
```

```
saldo =
[9734.05,12517.09,6304.24,11118.21,9542.27,2964.03,13232.54,11684.21,4919.
49,5293.05,8847.02,1789.03,11113.87,11665.19,2432.18,11626.03,10038.05,169
0.01,6409.31,13546.88,1298.90,1200.00,16000.00,5000.00]

dataframe = pd.DataFrame({"Nome":nome,"Idade":idade,"Saldo":saldo})
```

Vamos começar a construir o algoritmo de discretização. Para isso, vamos precisar de um método da biblioteca `sklearn` (*scikit-learn*) chamado `KBinsDiscretizer`. Vamos importá-lo com o código a seguir:

```
from sklearn.preprocessing import KBinsDiscretizer
```

Vamos instanciar nossa classe passando os seguintes parâmetros:

- `n_bins=4` para o número de intervalos que queremos criar — que, nesse caso, serão 4, para compararmos com os outros dois métodos;
- `encode="ordinal"`, indicado para codificar o resultado transformado — nesse caso, passaremos o ordinal para que o método retorne o identificador do intervalo como um valor inteiro;
- `strategy="kmeans"` para indicar ao algoritmo que desejamos fazer a discretização dos nossos dados via *k-means*.

```
discretizador =
KBinsDiscretizer(n_bins=4,encode="ordinal",strategy="kmeans")
```

Agora, podemos treinar nossa classe instanciada com o método `fit.`. Para isso, utilizaremos o conjunto que criamos no início desta seção. Desta forma, o método `KBinsDiscretizer` instanciado na variável `discretizador` consegue ajustar o modelo matemático aos dados passados como parâmetro:

```
discretizador.fit(dataframe["saldo"].to_frame())
```

Note que utilizamos `to_frame()` da biblioteca *pandas*, pois a biblioteca *scikit-learn* pede um pandas dataframe. Entretanto, como estamos passando uma única coluna, teremos uma *series* e por

causa disso teremos um erro. Ao passar `to_frame()`, essa única coluna será interpretada como um dataframe e não como uma *series*.

```
dataframe["saldo_discretizado"] =
discretizador.transform(dataframe["saldo"].to_frame())

dataframe.head()
```

O resultado da discretização utilizando *k-means* pode ser visto na seguinte tabela:

Nome	Idade	Saldo	saldo_discretizado
Miguel	75	9734.05	4.0
Arthur	42	12517.09	6.0
Heitor	50	6304.24	3.0
Helena	61	11118.21	5.0
Alice	77	9542.27	4.0

Podemos inspecionar os centros aprendidos pelo nosso discretizador com o comando `discretizador.bin_edges_`. Você deve notar que não utilizamos o comando `.bin_edges_` em nenhuma parte do livro até agora. Quando executado, esse comando mostra as bordas ou limites (*edges*) de cada um dos intervalos de discretização criado pelo *scikit-learn*.

```
discretizador.bin_edges_

array([array([ 1200.,1609.92,2467.21,4274.624,7562.78, 10490.92, 12270.16,
14549.41, 16000.])],dtype=object)
```

Vamos criar o gráfico com a função que criamos anteriormente para conseguirmos observar as proporções entre os intervalos criados:

```
proporcoes = dataframe['saldo_discretizado'].value_counts(normalize=True)
proporcoes.plot(kind='bar')
```

```
plt.title('Proporções do saldo discretizado')
plt.xlabel('Valor discretizado')
plt.ylabel('Proporção')
```

Agora podemos criar um gráfico utilizando código para podermos comparar as proporções entre os intervalos criados:

Figura 3.3: Proporção das observações entre os bins criados.

Com o gráfico criado, podemos observar que os intervalos de número `2.0` possui mais observações do que os demais e você pode estar se perguntando agora se isso é bom ou ruim. A resposta é: depende da *acurácia* do modelo, e esse é um assunto que veremos em mais detalhes em outro capítulo. Na próxima seção, veremos

como utilizar o processo de *árvores* para realizar a discretização do nosso conjunto.

3.2 Discretizando valores numéricos utilizando árvore e a biblioteca scikit-learn

A discretização que veremos agora também é um método definido como não supervisionado. Este método utiliza árvores de decisão para identificar os *bins* ideais nos quais classificaremos os valores variáveis.

A árvore de decisão é construída usando a variável (que vamos discretizar) e a variável-alvo (que é a que queremos predizer). Quando fazemos uma previsão, é atribuída uma observação a uma das `N` folhas finais. Portanto, qualquer árvore de decisão gerará uma saída discreta, cujos valores são as previsões em cada uma de suas `N` folhas.

A discretização com árvores de decisão cria uma relação monótona entre os *bins* que queremos discretizar e a variável-alvo (ou a variável que queremos prever).

> Relação monótona (ou monotônica) é uma relação entre variáveis na qual elas se movem na mesma direção, mas não necessariamente a uma taxa constante.

Vamos utilizar a mesma tabela do algoritmo de *k-means* e, da mesma forma, aqui não faremos os cálculos manualmente. Dessa vez, utilizaremos as bibliotecas *scikit-learn* e *feature-engine*. Vamos relembrar a tabela:

```
dataframe.head()
```

Nome	Saldo	saldo_discretizado
Miguel	9734.05	4.0
Arthur	12517.09	6.0
Heitor	6304.24	3.0
Helena	11118.21	5.0
Alice	9542.27	4.0

Vamos iniciar o nosso código para discretizar o nosso dataframe utilizando árvores. Para isso, vamos importar da biblioteca `sklearn` a classe `DecisionTreeRegressor`.

```
from sklearn.tree import DecisionTreeRegressor
arvore = DecisionTreeRegressor(max_depth=3, random_state=0)
```

A classe `DecisionTreeRegressor` deve receber os seguintes parâmetros:

- `max_depth`: profundidade máxima da árvore;
- `random_state`: parâmetro utilizado para garantir a reprodutibilidade do exemplo.

PARA QUE SERVE A REPRODUTIBILIDADE?

A maioria dos algoritmos de *machine learning* seleciona linhas de forma aleatória do dataframe. Sendo assim, utilizamos o parâmetro `random_state` para forçar o algoritmo de *machine learning* a selecionar sempre as mesmas linhas. Isso é útil para que não tenhamos variações no resultado final do nosso experimento.

Agora que criamos nossa árvore de decisão, é hora de treinarmos o nosso modelo. Entretanto, neste processo em específico, precisamos fornecer a variável que desejamos discretizar e a variável que

desejamos prever. Em nosso caso, desejamos discretizar a variável `idade` e usar a discretização para prever a variável `saldo`. A razão para usarmos a coluna `idade` como variável guia para a discretização do `saldo` por meio de um algoritmo não supervisionado é permitir que a máquina aprenda automaticamente os padrões subjacentes nos dados. Embora a `idade` possa parecer não estar relacionada à variável `saldo` em uma primeira vista, é possível que exista alguma relação implícita entre elas.

Então, ao fornecermos a `idade` como guia, permitimos que a árvore de decisão encontre agrupamentos ou padrões no conjunto de dados que podem influenciar a discretização do `saldo`. É importante destacar que os algoritmos não supervisionados, como a árvore de decisão, podem explorar relações ocultas nos dados sem a necessidade de rótulos ou respostas predefinidas. Dessa forma, ao utilizarmos a coluna `idade` para orientar a discretização do `saldo` com um algoritmo não supervisionado, estamos aproveitando ao máximo os dados disponíveis e permitindo que a máquina descubra possíveis associações entre essas variáveis, potencialmente revelando informações valiosas para a análise.

```
arvore.fit(dataframe[['Saldo']], dataframe['Idade'])
```

Note que aqui passamos o `dataframe[[Saldo']]` com dois colchetes, pois queremos que o *scikit-learn* interprete essa coluna como um dataframe e não como um *array*. Agora que treinamos nossa árvore, vamos utilizar o método `predict` para podermos discretizar, e o resultado será armazenado em uma nova coluna chamada `saldo_discretizado`:

```
dataframe['saldo_discretizado'] = arvore.predict(dataframe[['Saldo']])
```

Podemos utilizar o comando `round` para arredondar com duas casas decimais os valores da coluna `saldo_discretizado` e, em seguida, vamos exibir uma parte do nosso dataframe com o comando `dataframe.head()`.

Nome	Idade	Saldo	saldo_discretizado
Miguel	75	9734.05	69.17
Arthur	42	12517.09	42.0
Heitor	50	6304.24	57.17
Helena	61	11118.21	69.17
Alice	77	9542.27	69.17

Vemos que foram criados valores para cada saldo. Podemos averiguar quantos valores foram criados utilizando o comando `dataframe["saldo_discretizado"].unique()` . O resultado obtido é o seguinte:

```
array([69.17,42.,57.17,84.,26.5,87.,54.])
```

Embora não sejam valores inteiros, foram criados aqui 7 valores diferentes, sendo que cada um deles representa um *bin*. Nós podemos observar essas proporções construindo um gráfico:

```
contagem = dataframe['saldo_discretizado'].value_counts().sort_index()

plt.bar(contagem.index, contagem.values)
plt.xlabel('Saldo discretizado')
plt.ylabel('Contagem')
plt.title('Proporção')
```

O código que criamos é similar aos códigos dos gráficos de barras que criamos para mensurar a proporção de observações em cada um dos *bins* e pode ser visto na imagem a seguir:

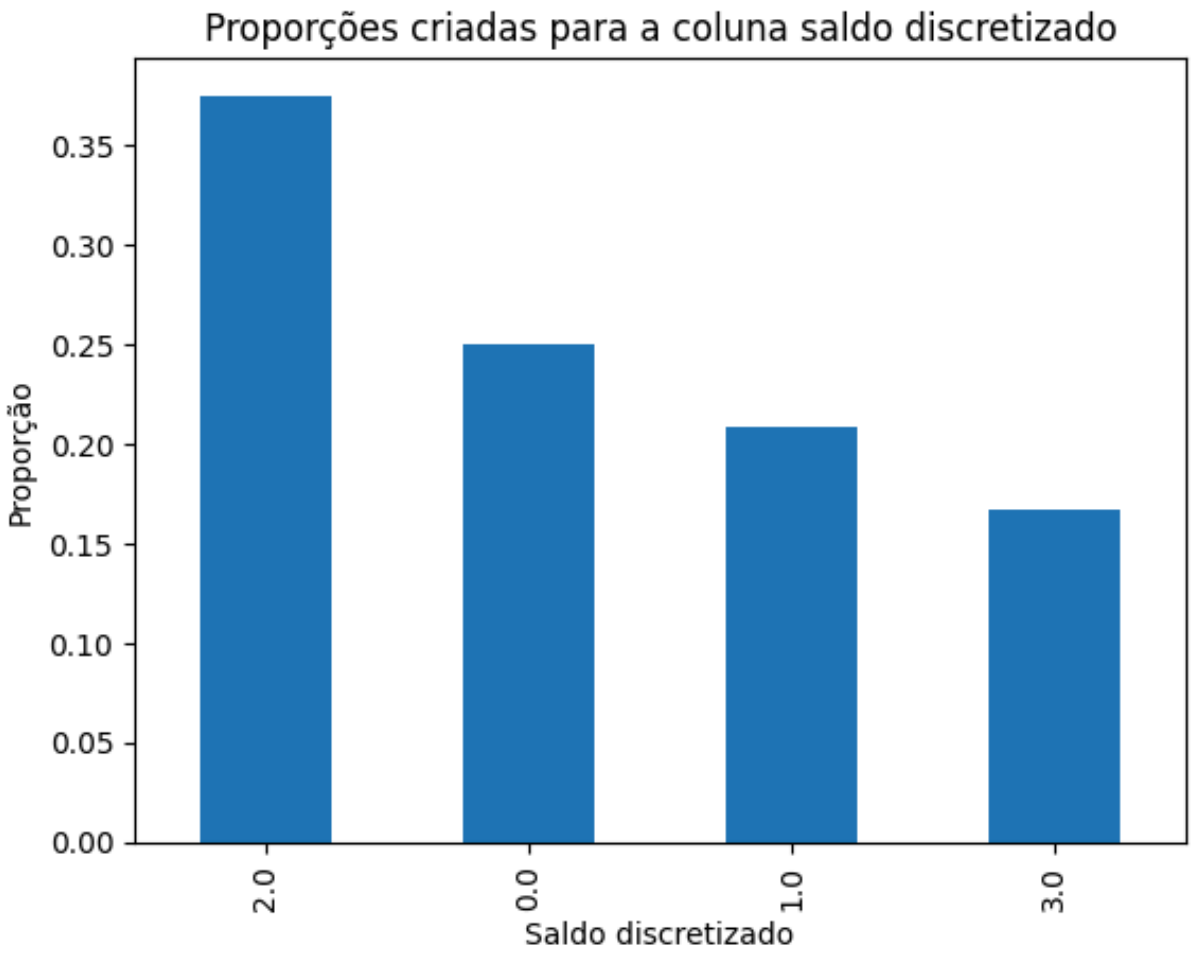

Figura 3.4: Resultado da discretização da variável idade.

Uma razão adicional para realizarmos esse tipo de discretização é aproveitar a monotonicidade das variáveis, o que pode ser útil ao lidar com algoritmos de *machine learning* linearmente dependentes. Esses algoritmos linearmente dependentes recebem esse nome porque dependem fortemente de uma relação linear entre as variáveis de entrada e de saída. Em outras palavras, eles são projetados para resolver problemas nos quais existe uma relação linear clara entre os dados utilizados.

Então, ao criarmos uma discretização baseada na monotonicidade das variáveis, estamos tornando o problema mais adequado para esses algoritmos. Ao estabelecermos uma relação monótona entre as variáveis, estamos simplificando o problema permitindo que os

algoritmos lineares identifiquem e explorem essa direção consistente de mudança entre as variáveis.

> **Linear ou monótona?**
>
> Uma relação monótona refere-se à direção consistente de mudança entre duas variáveis. Em uma relação monótona crescente, à medida que uma variável aumenta, a outra aumenta também. Em uma relação monótona decrescente, à medida que uma variável aumenta, a outra diminui.
>
> Já a relação linear é um tipo específico de relação monótona. Nela, à medida que uma variável aumenta (ou diminui), a outra também aumenta (ou diminui). Entretanto, vale ressaltar aqui que nem toda relação monótona é linear. Um exemplo disso são as relações quadráticas, na qual o aumento de uma variável resulta em um aumento inicial e, depois, em uma diminuição da outra variável — esse é um exemplo de relação monótona não linear.

Para mostrar a relação entre as variáveis `Saldo` e `Idade`, vamos criar o gráfico a seguir:

```
plt.figure(figsize=(12,7))
dataframe.groupby("Saldo")["Idade"].mean().plot()
plt.title("Relação não monótona entre as variáveis saldo e idade")
plt.ylabel("Idade");
```

E o resultado pode ser visto na imagem a seguir:

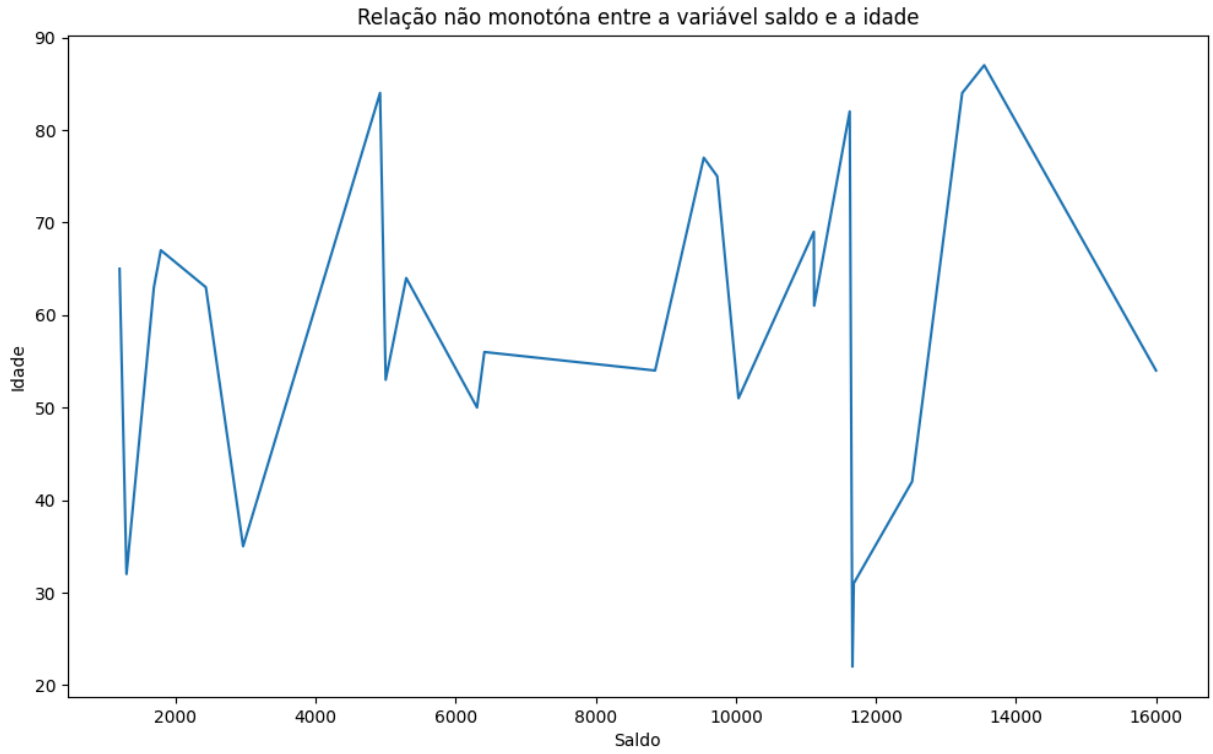

Figura 3.5: Resultado da discretização da variável idade.

Perceba que no gráfico anterior não temos uma relação de monotonicidade já que a variável `Idade` cresce e decresce à medida que variável saldo cresce.

Agora, vamos exibir a relação entre a variável `saldo discretizado` e `Idade` para que você perceba a diferença.

```
plt.figure(figsize=(12,7))
dataframe.groupby("saldo_discretizado")["Idade"].mean().plot()
plt.title("Relação não monótona entre a variável saldo e a idade")
plt.ylabel("Idade");
```

E o resultado pode ser visto no gráfico a seguir:

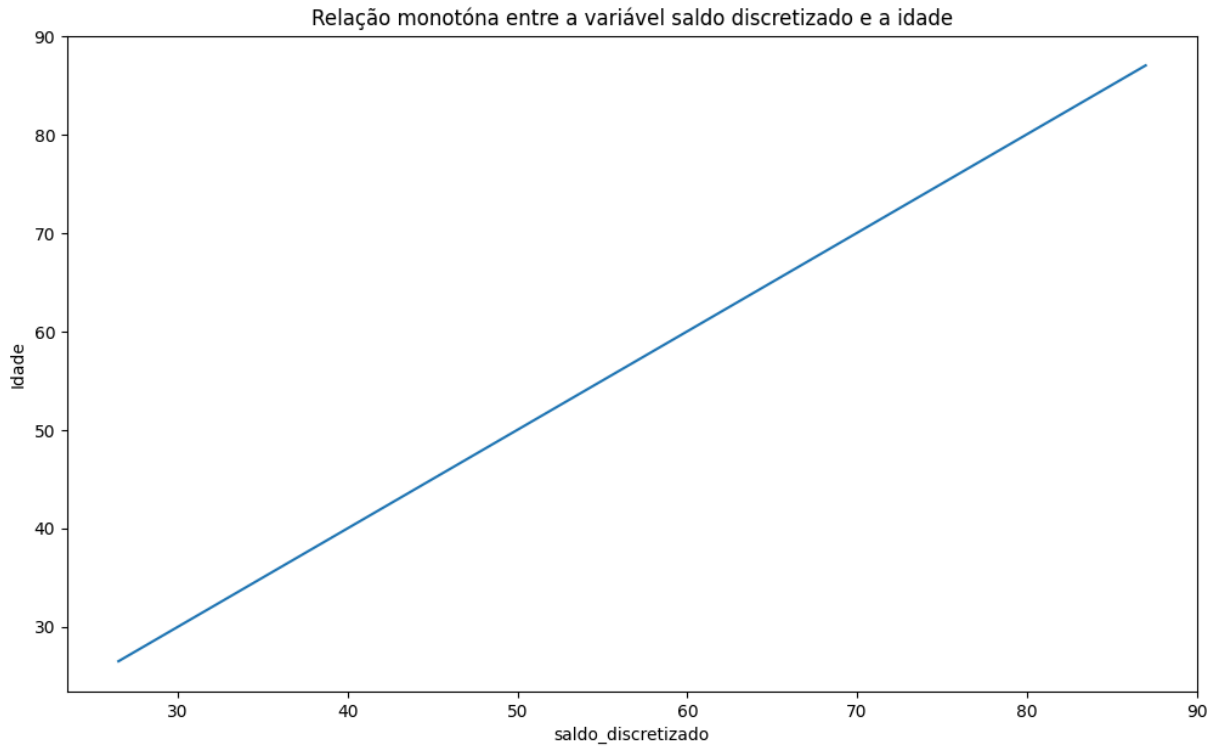

Figura 3.6: Resultado da discretização da variável idade.

Perceba que agora, sim, temos uma relação monótona entre as variáveis `saldo_discretizado` e `idade`, o que servirá caso utilizemos algoritmos de *machine learning* que são linearmente dependentes.

Na próxima parte, veremos como utilizar a biblioteca *feature-engine* para fazer a discretização do saldo; assim, poderemos observar a diferença entre as discretizações feitas por essas duas bibliotecas.

Discretizando valores numéricos utilizando árvore e a biblioteca feature-engine

Vamos desenvolver o mesmo processo de discretização do exemplo anterior, desta vez utilizando a biblioteca *feature-engine*. Vamos usar o mesmo dataframe do exemplo da seção passada:

```
import pandas as pd

nome =
```

```
["Miguel","Arthur","Heitor","Helena","Alice","Theo","Davi","Lilia","Laura"
,"Gabriel","Bernardo","Samuel","Valentina","Heloísa","Enzo","Pedro","Loren
zo","Sophia","Maria","Eduarda","Cecília","Júlia","Isabella","Benício"]
idade =
[75,42,50,61,77,35,84,31,84,64,54,67,69,22,63,82,51,63,56,87,32,65,54,53]
saldo =
[9734.05,12517.09,6304.24,11118.21,9542.27,2964.03,13232.54,11684.21,4919.
49,5293.05,8847.02,1789.03,11113.87,11665.19,2432.18,11626.03,10038.05,169
0.01,6409.31,13546.88,1298.90,1200.00,16000.00,5000.00]

dataframe= pd.DataFrame({"Nome":nome,"Idade":idade,"Saldo":saldo})
```

O próximo passo é criarmos um gráfico de linha para conseguirmos visualizar a relação entre o saldo e a média das idades. Para isso, utilizaremos o `groupby` para criarmos os grupos e, em seguida, criaremos o gráfico, como no código a seguir:

```
plt.figure(figsize=(12,7))
dataframe.groupby("Saldo")["idade"].mean().plot()
plt.title("Relação não monotóna entre a variável saldo e a idade.")
plt.ylabel("Idade")
```

O resultado pode ser visto na figura a seguir:

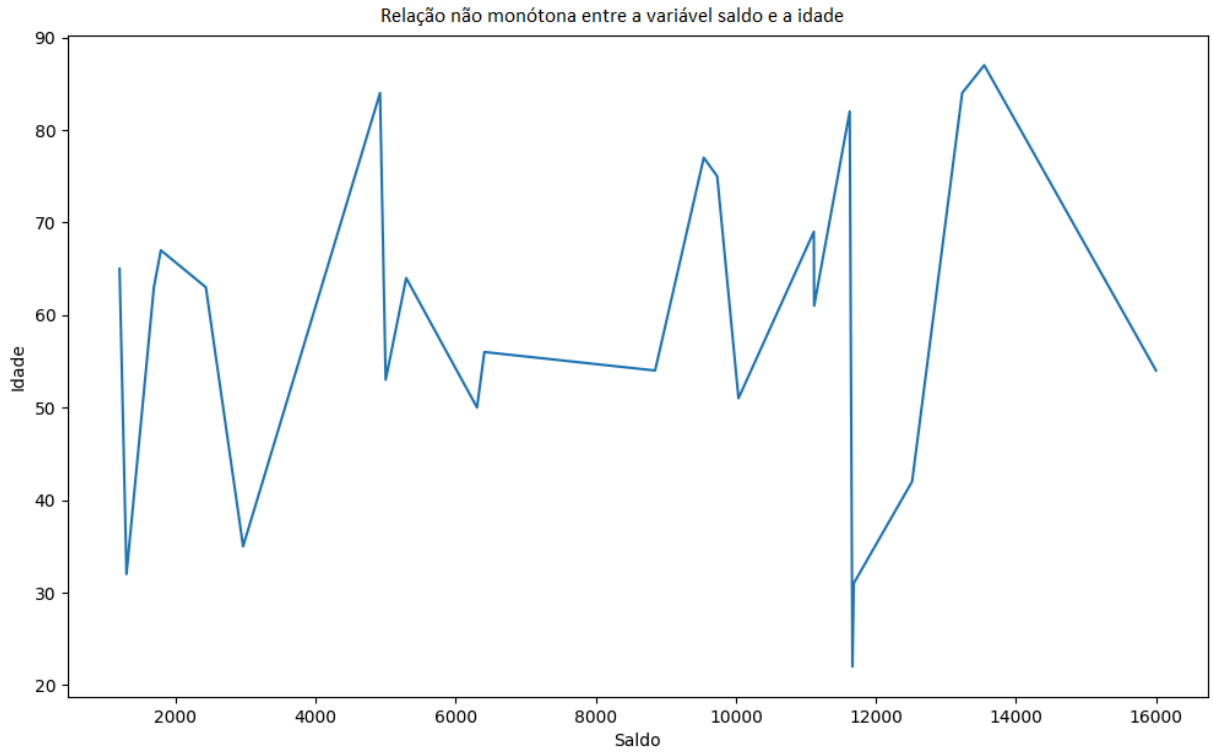

Figura 3.7: Relação não monótona entre a variável saldo e a idade.

Conseguimos perceber que o gráfico na figura anterior possui várias "quinas". Nosso trabalho agora é transformar esse gráfico em uma reta, ou no mais parecido possível com uma reta. Para isso, utilizaremos a classe `DecisionTreeDiscretizer` da biblioteca `feature_engine.discretisation`:

```
from feature_engine.discretisation import DecisionTreeDiscretiser
```

Depois de importada a classe `DecisionTreeDiscretiser` da biblioteca `feature_engine.discretisation`, vamos instanciar a nossa classe importada, da seguinte forma:

```
arvore_ft = DecisionTreeDiscretiser(cv=10,param_grid={"max_depth":[1,2]})
```

Vamos entender os parâmetros utilizados no código anterior:

- `cv`: refere-se ao número de *dobras* a serem usadas na *validação cruzada* durante o treinamento do modelo. A validação cruzada é uma técnica que divide o conjunto de dados

em subconjuntos para avaliar o desempenho do modelo de uma forma mais robusta. Nesse caso em particular, utilizamos o valor igual a 10 indicando assim que serão feitas 10 dobras.

- `max_depth` : representa a profundidade máxima da árvore de decisão. A profundidade de uma árvore refere-se ao número de níveis de decisão que ela possui, ou seja, quantas perguntas ou condições podem ser feitas para chegar a uma decisão final. Limitar a profundidade máxima da árvore pode nos ajudar com o controle da complexidade e evitar um ajuste excessivo aos dados de treinamento. Em nosso exemplo, utilizamos uma lista com os valores `[1,2]`, indicando que serão testadas duas profundidades diferentes em nossa árvore: 1 e 2.

O QUE SÃO ESSAS DOBRAS?

No aprendizado de máquina, o conceito de dobras, também conhecido como *folds*, refere-se à divisão do conjunto de dados em subconjuntos para realizar a validação cruzada. A validação cruzada é uma técnica usada para avaliar a eficácia de um modelo de *machine learning* de forma robusta.

Ao realizarmos a validação cruzada com um número específico de dobras, o conjunto de dados é dividido em partes iguais, chamadas dobras. Em cada iteração do processo, uma das dobras é usada como conjunto de teste, enquanto as demais dobras são usadas como conjunto de treinamento. Essa abordagem permite que o modelo seja testado em diferentes subconjuntos dos dados, garantindo uma avaliação mais precisa de sua capacidade de generalização.

O valor fornecido para o parâmetro `cv` indica o número de dobras que serão utilizadas na validação cruzada. Por exemplo, caso o valor de `cv` seja definido como 10, como em nosso exemplo, o conjunto de dados será dividido em 10 dobras ou subconjuntos. O modelo será treinado e testado 10 vezes, cada vez usando uma dobra diferente como conjunto de teste.

Prosseguindo, vamos criar a discretização utilizando a variável `saldo`. Da mesma forma como no exemplo anterior, precisamos passar a variável-alvo, que neste caso é a `idade`:

```
arvore_ft.fit(dataframe["saldo"].to_frame(),dataframe["idade"])
```

Com isso, conseguimos treinar a nossa árvore. Podemos então discretizar a variável `saldo`; porém, não utilizaremos aqui o método `fit`, mas, sim, o método `predict` para conseguirmos predizer os valores da variável `saldo`:

```
dataframe["saldo_discretizado"] =
arvore.predict(dataframe["saldo"].to_frame())
dataframe.head()
```

O resultado desse processo pode ser visto na tabela a seguir:

index	Nome	Idade	Saldo	saldo_discretizado
0	Miguel	75	9734.05	69.16
1	Arthur	42	12517.09	42.0
2	Heitor	50	6304.24	57.16
3	Helena	61	11118.21	69.16
4	Alice	77	9542.27	69.16

Podemos ver os valores criados na coluna `saldo_discretizado` utilizando o comando `unique()`:

```
dataframe["saldo_discretizado"].unique()
```

Após executar o código anterior, temos o seguinte resultado:

```
array([69.16666667,42.,57.16666667,84., 26.5,87., 54.])
```

Perceba que temos valores com valores decimais em nossa coluna saldo discretizado. Isso acontece para criarmos a relação monótona que desejamos, o que pode ser visto no gráfico a seguir:

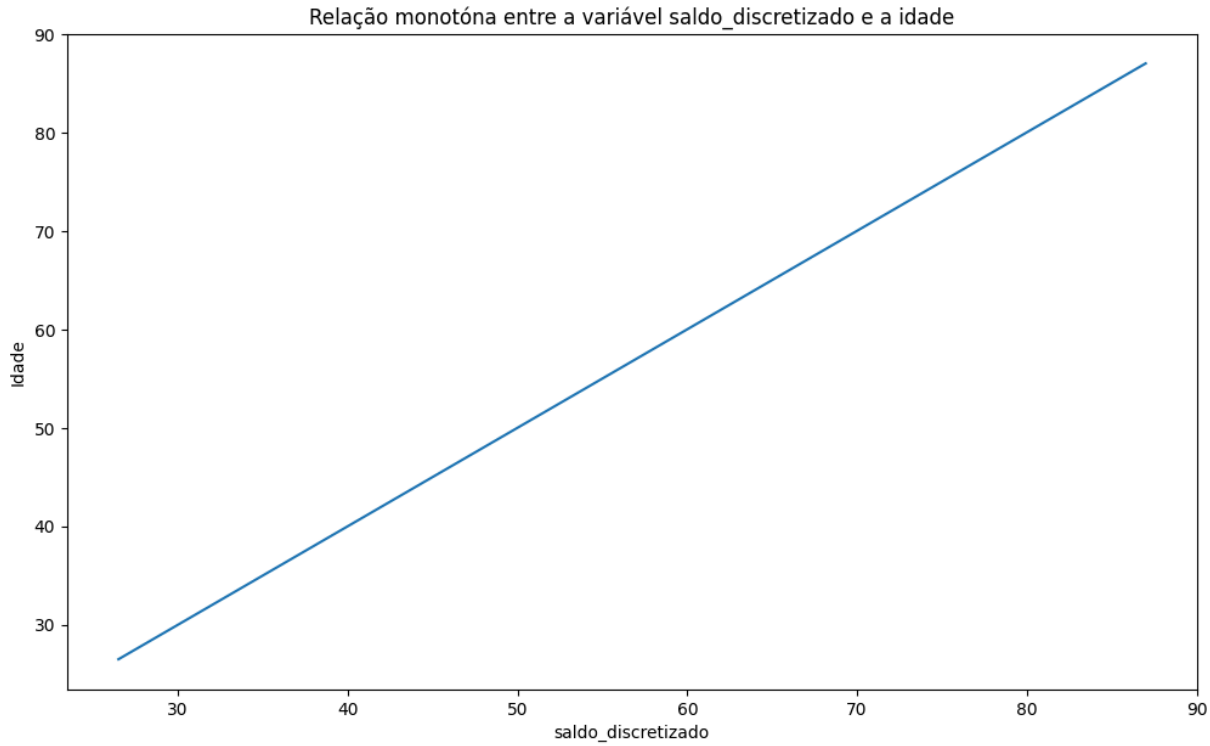

Figura 3.8: Relação monótona entre a variável saldo_discretizado e a média das idades.

Analisando o gráfico, conseguimos perceber que agora a relação entre a variável `saldo_discretizado` e a média das idades é uma relação monótona como queríamos.

Neste capítulo, vimos como discretizar valores numéricos. Embora pareça pouco, essas discretizações são importantes para melhorar o desempenho e até a velocidade de processamento de nossos algoritmos de *machine learning*.

No próximo capítulo, continuaremos com as discretizações, porém veremos como fazê-las em um conjunto maior de dados, utilizando dados de treino e dados de teste. Você conseguirá ver como esses algoritmos funcionam quando lidamos com volumes maiores de dados.

Capítulo 4
Projeto prático — Discretização de valores numéricos

Para o nosso segundo projeto prático, continuaremos trabalhando com o conjunto do Titanic. Porém, desta vez vamos nos concentrar nos valores numéricos das colunas `Age` e `Fare`. Neste projeto, vamos fazer a discretização dessas duas colunas utilizando as bibliotecas *scikit-learn* e *feature-engine*.

PassengerId	Survived	Pclass	Name	Sex	Age
892	0	3	Wilkes, James	male	34.5
893	1	3	Kelly, James	female	47.0
894	0	2	Thomas Francis	male	62.0
895	0	3	Albert	male	27.0
896	1	3	Alexander	female	22.0

PassengerId	SibSp	Parch	Ticket	Fare	Embark
892	0	0.0	330911	7.8292	Q
893	1	0.0	363272	7.0	S
894	0	0.0	240276	9.6875	Q
895	0	0.0	315154	8.6625	S
896	1	0.0	3101298	12.2875	S

Observação: Para este projeto prático, é importante utilizar o dataframe concluído no projeto anterior ou baixar o arquivo `projeto2.csv`, disponibilizado no seguinte link: https://github.com/ThiagoBenevides/feature-engine-transformando-dados-brutos-em-recursos/tree/main/datasets.

Os algoritmos que utilizaremos neste projeto não trabalham com valores ausentes.

Vamos dividir o nosso conjunto em um conjunto de treino e outro de teste, como fizemos no projeto prático anterior. Para isso, utilizaremos o seguinte código:

```
from sklearn.model_selection import train_test_split
X_treino, X_teste, y_treino, y_teste =
train_test_split(dataframe.drop("Survived",axis=1),dataframe["Survived"],t
est_size=0.3,random_state=0)
```

O primeiro dataframe apresenta os valores de treino e o segundo, os valores de teste do nosso conjunto, `X_treino` e `X_teste`, respectivamente. O resultado que você poderá ver se digitar os comandos `X_treino.head()` e `X_teste.head()` serão dois dataframes com todas as colunas, exceto a coluna de `Survived`, que está armazenada nas variáveis `y_treino` e `y_teste`.

Outra coisa que você pode perceber utilizando o comando `X_treino.shape` e `X_teste.shape` é que o dataframe `X_treino` possui mais linhas que o `X_teste`. Mais especificamente, o dataframe `X_treino` possui 70% dos dados do nosso dataframe original, e isso se deve ao parâmetro `test_size=0.3`, que nós inserimos quando fizemos a divisão dos nossos dados em um conjunto de treino e conjunto de teste.

Agora que preparamos tudo, vamos iniciar a discretização das colunas `Age` e `Fare` com as bibliotecas *scikit-learn* e a *feature-engine*.

4.1 Discretizando valores numéricos utilizando a biblioteca scikit-learn

A primeira discretização que veremos utilizando a biblioteca *scikit-learn* é a discretização de largura igual. Vamos importar a biblioteca que vamos utilizar:

```
from sklearn.preprocessing import KBinsDiscretizer
```

Vamos instanciar essa classe com os seguintes parâmetros:

- `n_bins`: para informarmos o número de *bins* que queremos criar — no nosso caso, serão 8;
- `encode`: método usado para codificar o resultado transformado — no nosso caso, será o `"ordinal"` para que o retorno seja o identificador `bin` codificado como um valor inteiro;
- `strategy`: o parâmetro `strategy` diz respeito à estratégia que o algoritmo usará para fazer a discretização dos dados. Neste caso em particular, queremos fazer a discretização uniforme dos dados, ou seja, ter os intervalos com um número similar de amostras, e por isso usaremos `uniform` no parâmetro `strategy`.

Vamos ao código:

```
discretizador =
KBinsDiscretizer(n_bins=8,encode="ordinal",strategy="uniform")
```

Agora que instanciamos nossa classe `KBinsDiscretizer`, podemos ajustar o modelo aos nossos dados de treino utilizando o comando `.fit()` como no código a seguir:

```
discretizador.fit(X_treino[["Age","Fare"]])
```

Perceba que, ao utilizar o método `.fit()`, passamos apenas as colunas que queremos discretizar, ou seja, a coluna `"Age"` e a coluna `"Fare"`. O resultado pode ser visto com o código a seguir:

```
discretizador.fit(X_treino[["Age","Fare"]])
KBinsDiscretizer(encode='ordinal',n_bins=8,strategy='uniform')
```

Após ajustarmos o modelo com os nossos dados de treino, podemos inspecionar os valores que o `KBinsDiscretizer` gerou para os limites superior e inferior das colunas `Age` e `Fare` com o comando `discretizador.bin_edges_`.

```
discretizador.bin_edges_
```

O resultado desse comando pode ser visto a seguir:

```
array([array(0.17,9.64875,19.1275,28.60625,38.085,47.56375,57.0425,66.5212
5,76.]),

array([0.,64.04115,128.0823,192.12345,256.16.46,320,20575,384.2469,448,288
05,512.3292])], dtype=object)
```

Como você pode observar, o comando `discretizador.bin_edges_` retornou para nós um *array* contendo dois *arrays*. Isso acontece pois estamos discretizando duas colunas, a coluna `Age` e a coluna `Fare`; sendo assim, o primeiro *array* se refere aos limites para a discretização da coluna `Age` e o segundo, aos limites para discretização da coluna `Fare`.

Com o modelo ajustado e os valores inspecionados, vamos transformar os nossos dados com o método `transform()`, como no

código a seguir:

```
treino_disc= discretizador.transform(X_treino)
teste_disc= discretizador.transform(X_teste)
```

> **LEMBRE-SE:** a biblioteca *scikit-learn* retorna um *numpy array* em vez de um dataframe. Portanto, devemos converter novamente em um dataframe caso desejemos utilizar os métodos da *pandas* em nosso conjunto de dados.

Para podermos criar os gráficos que mostram a proporção das observações em cada intervalo, vamos converter em um pandas dataframe nossos `treino_disc` e o `teste_disc`, da seguinte forma:

```
treino_disc = pd.DataFrame(treino_disc,columns=["Age","Fare"])
teste_disc = pd.DataFrame(teste_disc,columns=["Age","Fare"])
```

Criaremos a seguir um gráfico apresentando a proporção de observações em cada intervalo. O que podemos notar com isso é que, pelo fato de termos dois conjuntos, poderemos comparar as proporções nos dados de treino.

Vamos começar construindo um gráfico para apresentar a proporção da coluna `Age`. No código a seguir, fazemos uma proporção para os dados de treino e outra para os dados de teste.

```
df1 = treino_disc['Age'].value_counts() / len(treino_disc)
df2 = teste_disc['Age'].value_counts() / len(teste_disc)
```

O resultado desse trecho de código é a quantidade de cada intervalo discretizado dado em porcentagem. Então, para o `df1`, temos a seguinte distribuição:

```
|2.0|0.462329|
|3.0|0.160959|
|4.0|0.109589|
|1.0|0.099315|
```

```
|5.0|0.068493|
|0.0|0.054795|
|6.0|0.041096|
|7.0|0.003425|
```

E para o `df2` temos a seguinte distribuição:

```
|2.0|0.515873|
|3.0|0.166667|
|4.0|0.103175|
|1.0|0.095238|
|5.0|0.063492|
|0.0|0.031746|
|6.0|0.015873|
|7.0|0.007937|
```

Continuando o código, a próxima parte vai unir o `df1` e o `df2` em um único dataframe, que armazenaremos na variável `aux`. Para isso, utilizaremos o comando `concat`:

```
aux = pd.concat([df1, df2], axis=1)
```

O resultado dessa concatenação é:

	Age	**Age**
2.0	0.462329	0.515873
3.0	0.160959	0.166667
4.0	0.109589	0.103175
1.0	0.099315	0.095238
5.0	0.068493	0.063492
0.0	0.054795	0.031746
6.0	0.041096	0.015873
7.0	0.003425	0.007937

A coluna à esquerda pertence ao dataframe `df1` e a coluna à direita pertence ao `df2`. Então vamos renomear as colunas para `treino` e `teste` para conseguirmos construir os gráficos com as proporções de cada um dos intervalos da discretização:

```
aux.columns = ['treino','teste']
```

E agora vamos construir o gráfico de barra exibindo a proporção de cada um dos intervalos para a coluna de treino e a coluna de teste, conforme o código a seguir:

```
aux.plot.bar()
plt.xticks(rotation=45)
plt.ylabel('Número de observações por intervalos')
```

O resultado pode ser visto na figura a seguir:

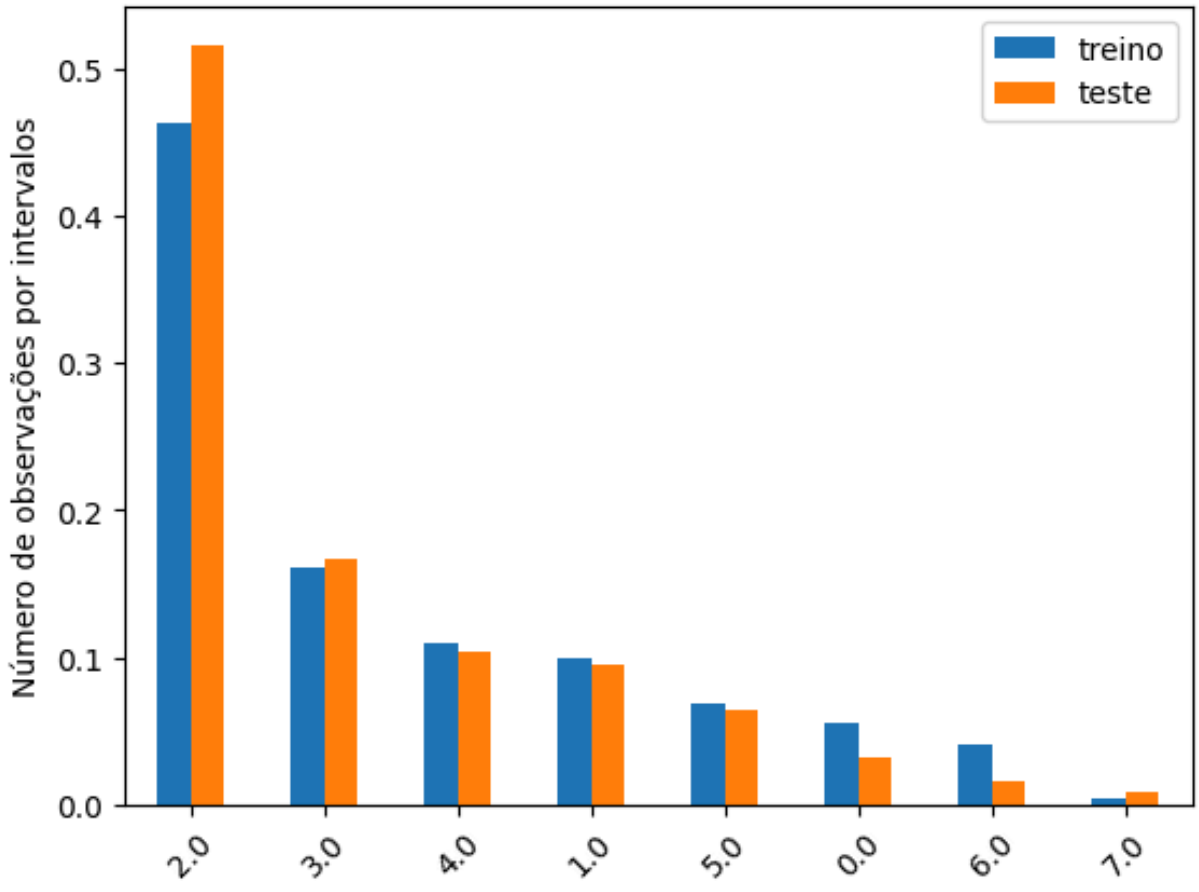

Figura 4.1: Número de observações por intervalos.

Com esse gráfico, conseguimos visualizar mais facilmente que a discretização que fizemos ficou desproporcional. Por exemplo, conseguimos ver no gráfico que o intervalo `2.0`, que são as duas primeiras barras da esquerda para a direita, possui uma quantidade maior de observações se compararmos aos demais intervalos.

Vamos criar o mesmo gráfico, porém desta vez utilizaremos a coluna `Fare`:

```
df1_fare = treino_disc['Fare'].value_counts() / len(treino_disc)
df2_fare = teste_disc['Fare'].value_counts() / len(teste_disc)

aux_fare = pd.concat([df1_fare, df2_fare], axis=1)
aux.columns = ['treino','teste']

aux_fare.plot.bar()
plt.xticks(rotation=45)
plt.ylabel('Número de observações por intervalos')
```

Veja que o código é similar ao utilizado anteriormente, sendo que as únicas coisas que mudaram foram os nomes das variáveis e da coluna, que agora é `Fare`, em vez de `Age`. O resultado pode ser visto na figura a seguir:

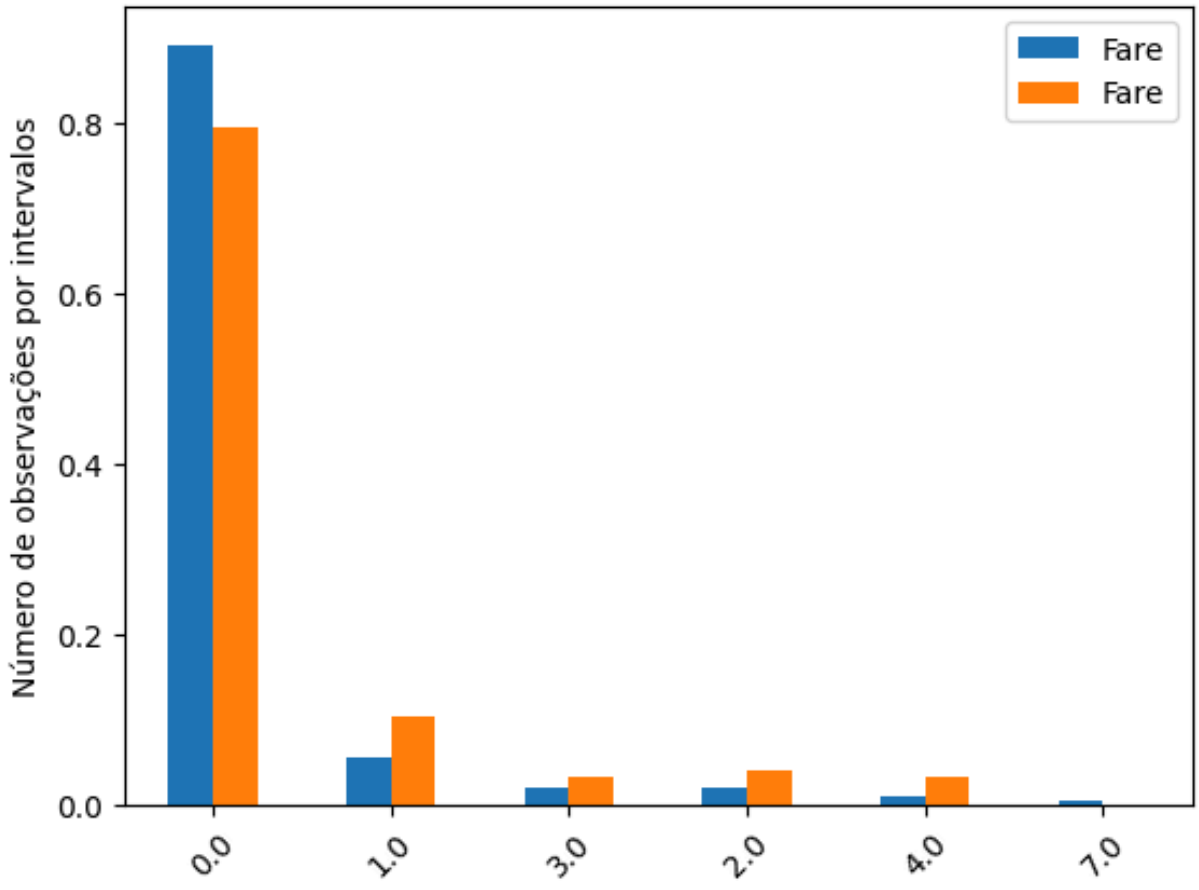

Figura 4.2: Número de observações da coluna Fare.

Olhando novamente para o gráfico anterior a este, vemos que os intervalos foram criados de maneira desbalanceada, isto é, cada um deles possui uma quantia superior do que a dos demais — por isso, o intervalo marcado como `2.0` ficou com uma quantidade maior.

O que é desproporcional neste caso agora é a quantidade de observações em cada um dos intervalos. Observe a coluna que representa o primeiro intervalo da esquerda para direita. Perceba que as duas barras que representam a quantidade de observações desse primeiro intervalo são maiores do que a dos demais intervalos. Essas distorções em nossa discretização podem provocar erros nos algoritmos de *machine learning*, fazendo com que eles tenham uma baixa acurácia.

Na próxima seção, faremos a mesma discretização, mas desta vez utilizando a biblioteca *feature-engine*.

4.2 Discretizando valores numéricos utilizando a biblioteca feature-engine

Se você estiver utilizando o *Google Colaboratory*, lembre-se de que é necessário reinstalar a biblioteca *feature-engine* sempre que precisar usá-la ou toda vez que o ambiente for resetado, caso contrário, você receberá um erro dizendo que a biblioteca não foi instalada. Depois de instalá-la, podemos importar a classe que vamos utilizar nesta seção, a `EqualWidthDiscretiser` :

```
from feature_engine.discretisation import EqualWidthDiscretiser
```

Vamos instanciar a nossa classe `EqualWidthDiscretiser` com os seguintes parâmetros:

- `bins` , para informar ao método qual é o número de intervalos que queremos criar;
- `variables` , para informar ao método quais colunas vamos discretizar.

Com o código a seguir, vamos instanciar nossa classe:

```
discretizador_ft = EqualWidthDiscretiser(bins=8, variables=["Age","Fare"])
```

Podemos treinar o nosso discretizador com o comando `.fit()` passando o conjunto de dados de treino, como no código a seguir:

```
discretizador_ft.fit(X_treino)
```

Podemos inspecionar os limites dos intervalos criados com o comando `discretizador_ft.binner_dict` . O resultado pode ser visto a

seguir:

```
{'Age': [-
inf,9.64875,19.1275,28.606250000000003,38.085,47.56375,57.042500000000004,
66.52125,inf],
 'Fare': [-
inf,64.04115,128.0823,192.12345,256.1646,320.20575,384.2469,448.28805,inf]
}
```

Uma coisa que você pode notar é que tanto em `Age` quanto em `Fare` temos o `-inf` e o `inf` aparecendo. Isso serve para que os valores que são considerados outliers sejam encaixados no primeiro ou no último intervalo.

O QUE SÃO OUTLIERS?

Em estatística, outliers são observações que se desviam significativamente do padrão ou do comportamento esperado de um conjunto de dados. São valores extremos que estão muito distantes dos demais pontos de dados. Essas observações são consideradas "fora da curva" ou "pontos fora do comum" em relação ao restante dos dados.

Os outliers podem surgir de várias maneiras. Podem ser resultados de erros de medição, erros de entrada de dados ou até mesmo representar eventos raros ou incomuns em um determinado contexto. Podem ocorrer também devido a causas naturais, como variabilidade inerente a certos fenômenos ou características.

Identificar e tratar os *outliers* é importante em análises estatísticas, pois podem distorcer a interpretação e os resultados obtidos. Caso esses valores não sejam tratados de forma adequada, podemos ter conclusões errôneas e influenciar negativamente nossas análises estatísticas, como a média, a variância e a regressão. No entanto, é importante ressaltar que nem todos os valores discrepantes são necessariamente outliers. Em alguns casos, eles podem representar informações válidas e importantes. Portanto, é essencial entender o contexto dos dados e as características específicas do problema em análise antes de classificar um ponto como *outlier*.

Vamos transformar nossos conjuntos de treino e teste e armazená-los em outras variáveis, assim como fizemos com a biblioteca *scikit-learn*:

```
df_treino = discretizador_ft.transform(X_treino)
df_teste = discretizador_ft.transform(X_teste)
```

O resultado pode ser visto com os comandos `df_treino.head()` e `df_teste.head()`. Ao utilizar esses comandos, você notará que o dataframe retornado é similar ao nosso primeiro dataframe com os conjuntos de treino e teste. Porém, você notará que as colunas `Age` e `Fare` tiveram seus valores alterados para o número dos respectivos intervalos.

Com isso, conseguimos perceber a seguinte diferença entre as bibliotecas *scikit-learn* e *feature-engine*:

- *Scikit-learn*: retorna para nós um `numpy.array`;
- *Feature-engine*: retorna para nós um `pandas.DataFrame.DataFrame`.

Agora, vamos exibir a proporção das observações da variável `Age` e, para isso, utilizaremos o código a seguir:

```
df1_disc = df_treino['Age'].value_counts() / len(df_treino)
df2_disc = df_treino['Age'].value_counts() / len(df_teste)

aux_disc = pd.concat([df1_disc, df2_disc], axis=1)

aux_disc.columns = ['treino','teste']

aux_disc.plot.bar()
plt.xticks(rotation=45)
plt.ylabel('Número de observações por intervalos');
```

O resultado desse código pode ser visto a seguir:

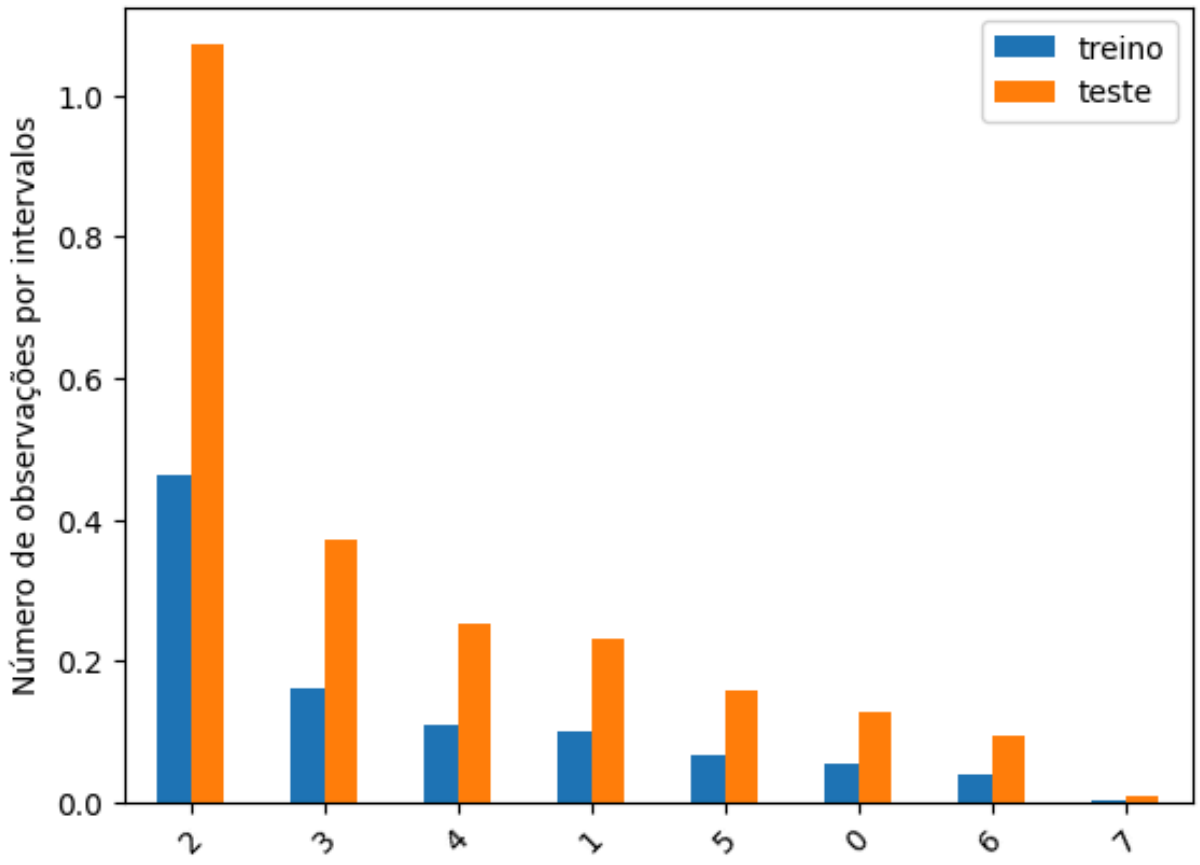

Figura 4.3: Proporção entre as observações da variável Age.

Inicialmente, podemos ser levados a pensar que não houve uma mudança significativa. Entretanto, ao observar com atenção, podemos identificar pequenas diferenças entre a discretização utilizando a biblioteca *scikit-learn* e a biblioteca *feature-engine*. Uma dessas diferenças pode ser observada na discretização da coluna `Age`, especificamente na variável de rótulo `2`, na qual é possível notar uma variação nos valores entre os conjuntos de treino e teste. Essa divergência ocorre devido às diferentes abordagens e cálculos realizados por essas duas bibliotecas, o que pode resultar em erros de arredondamento e truncamento dos números.

Diante disso, surge a pergunta: *qual é a melhor biblioteca?* A resposta depende dos testes que podem ser realizados para calcular o erro, o que será discutido no apêndice *Qual é o melhor modelo?*, onde abordaremos os testes e a avaliação da qualidade das nossas

transformações. Podemos observar variações também na discretização da coluna `Fare` ; para isso, utilizaremos o código a seguir:

```
df1_disc_fare = df_treino['Fare'].value_counts() / len(df_treino)
df2_disc_fare = df_treino['Fare'].value_counts() / len(df_teste)

aux_disc_fare = pd.concat([df1_disc_fare, df2_disc_fare], axis=1)

aux_disc_fare.columns = ['treino','teste']

aux_disc_fare.plot.bar()
plt.xticks(rotation=45)
plt.ylabel('Número de observações por intervalos');
```

E o resultado ao executarmos esse código anterior é similar à figura a seguir:

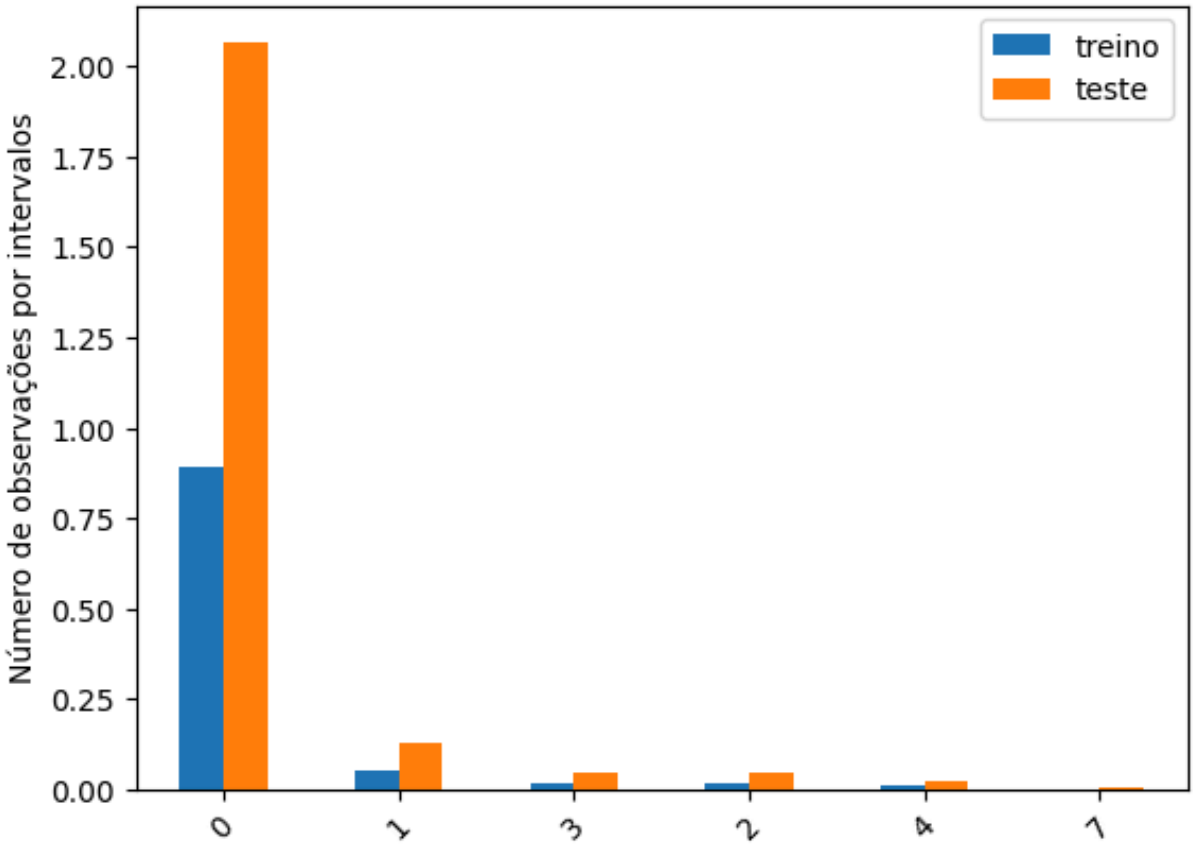

Figura 4.4: Número de observações por intervalo criado.

Assim, conseguimos ver que, embora utilizemos outra biblioteca, os intervalos criados não diferem muito, ou seja, a distorção provocada pela falta de proporção entre o número de observações persiste. Então, na próxima seção, veremos como fazer a discretização por frequência igual utilizando os quantis.

4.3 Discretização de valores numéricos utilizando faixa de frequência igual com a biblioteca scikit-learn

Vamos criar a discretização de frequência igual utilizando a biblioteca *scikit-learn* com o mesmo conjunto de dados. Para isso, vamos

começar importando a biblioteca `train_test_split` da biblioteca `sklearn` para fazermos a divisão em dados de treino e dados de teste:

```
from sklearn.model_selection import train_test_split
X_treino, X_teste, y_treino, y_teste =
train_test_split(dataframe.drop("Survived",axis=1),dataframe["Survived"],t
est_size=0.3,random_state=0)
```

Vamos importar a biblioteca `KBinsDiscretizer` , a mesma que utilizamos para fazer a discretização de largura igual, porém, como queremos dividir por frequência igual, vamos passar no parâmetro `strategy` o `quantile` , para fazermos a divisão baseada em quantis:

```
discretizador_freq = KBinsDiscretizer(n_bins=4,
encode="ordinal",strategy="quantile")
```

Podemos agora treinar nosso discretizador com o conjunto de dados de treino:

```
discretizador_freq.fit(X_treino)[["Age","Fare"]]
```

Vamos inspecionar os valores dos limites do intervalo calculado com o comando `discretizador_freq.bin_edges_` . E o retorno será o seguinte:

```
array([array([0.17,0.18,21.,25.27.,28.,32.,39.8,48.9,76.]),
       array([0., 7.65,
7.7958,8.05,10.5,13.8604,20.445,26.,36275,69.095,512.3292])],dtype=object)
```

Vamos transformar nossos dados e armazená-los em uma nova variável, como fizemos com a discretização passada, e, em seguida, converter os dados transformados em dataframe para podermos fazer os gráficos de comparação:

```
treino_freq = discretizador_freq.transform(X_treino[["Age","Fare"]])
teste_freq = discretizador_freq.transform(X_teste[["Age","Fare"]])

treino_freq = pd.DataFrame(treino_freq,columns=["Age","Fare"])
teste_freq = pd.DataFrame(teste_freq,columns=["Age","Fare"])
```

Com os dataframes em mãos, vamos criar os gráficos para comparação. Pelo fato de o algoritmo que vamos construir ser semelhante ao da discretização passada, criaremos uma função para esse código. Assim, sempre que quisermos construir um gráfico de comparação entre as proporções, podemos simplesmente chamar a função.

A função que faremos não é uma função elaborada e serve apenas para fazermos a comparação de duas proporções, então ela vai receber o conjunto de dados de treino e *plotar* o gráfico de barras.

```
def compara_proporcao(dataframe1,dataframe2)
    prop1 = dataframe1.value_counts() / len(dataframe1)
    prop2 = dataframe2.value_counts() / len(dataframe2)

    aux = pd.concat([prop1, prop2], axis=1)
    plt.figure(figsize=(12,9))
    aux.columns = ['treino', 'teste']
    aux.plot.bar()
    plt.xticks(rotation=45)
    plt.ylabel('Número de observações por intervalos')
```

Agora, vamos chamar a função para compararmos as idades:

```
compara_proporcao(treino_freq["Age"],teste_freq["Age"])
```

Temos como resultado o que é apresentado na imagem a seguir:

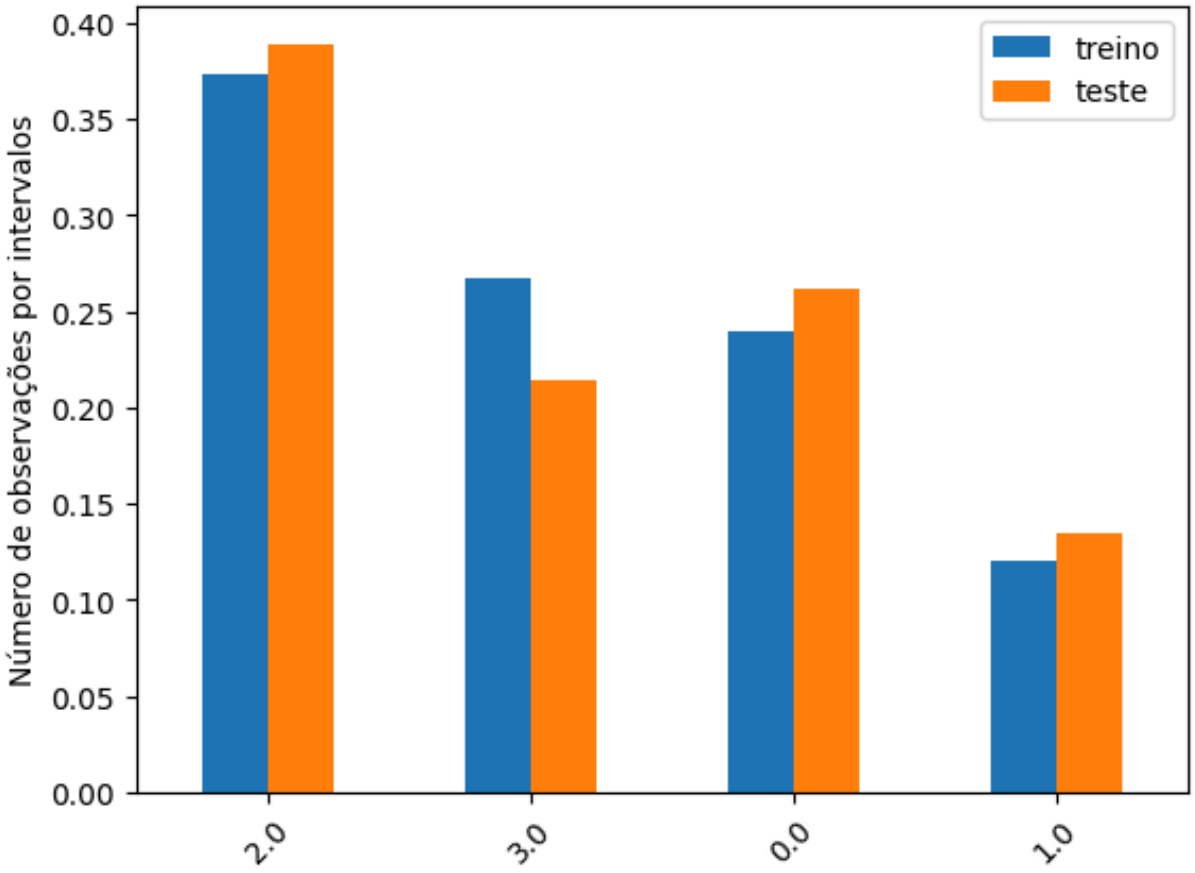

Figura 4.5: Comparando proporções entre os intervalos criados.

Podemos observar pelo gráfico que os intervalos são criados de uma forma mais proporcional do que antes. Vamos utilizar a função criada para compararmos agora a coluna `Fare`.

```
compara_proporcao(treino_freq["Fare"],teste_freq["Fare"])
```

O resultado pode ser visto na imagem a seguir:

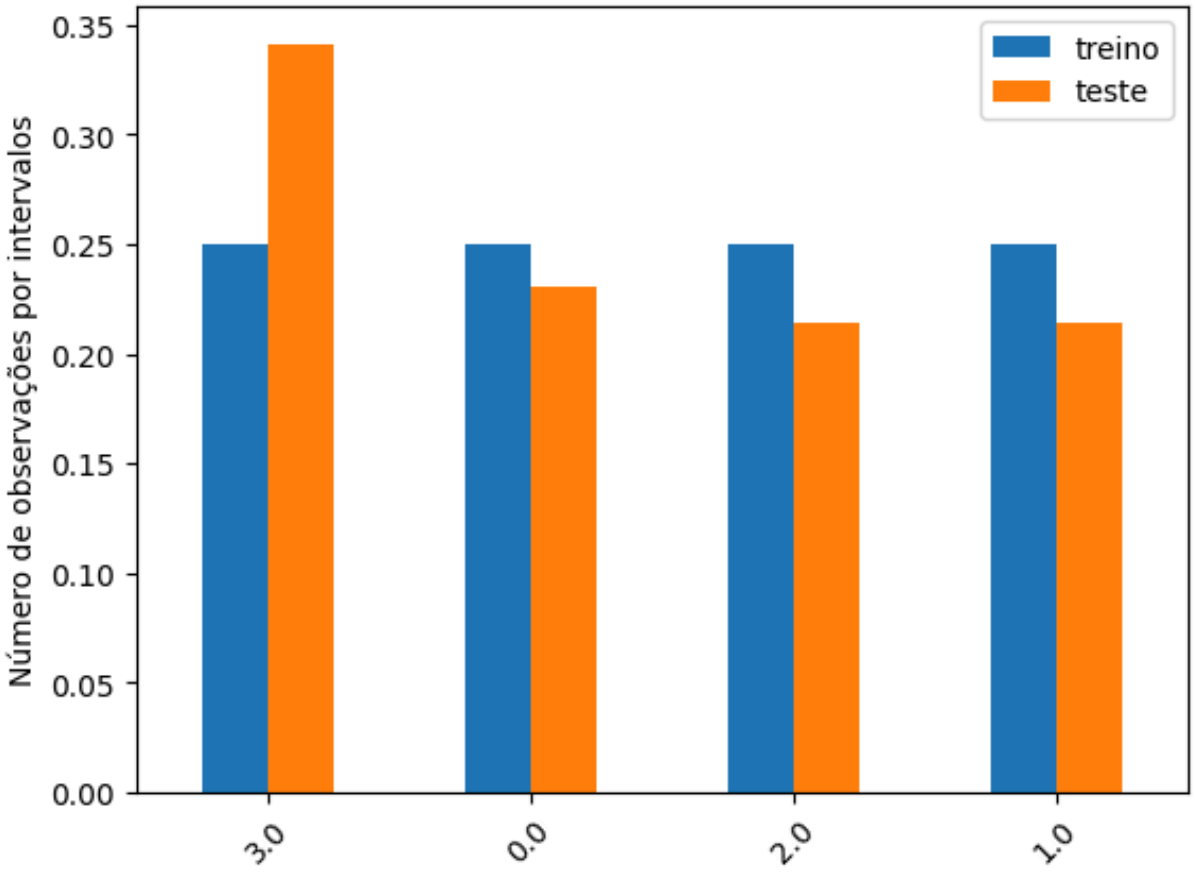

Figura 4.6: Comparação entre as proporções de observações por intervalo da coluna Fare.

Da mesma forma como a coluna `Age`, temos um intervalo que possui mais observações no conjunto de teste e treino, que é o conjunto de número `3.0`. Entretanto, se compararmos a proporção da coluna `Fare` usando largura de faixa e frequência igual, veremos que a discretização por frequência conseguiu deixar os intervalos com uma proporção melhor do que a de largura igual.

Com essas duas comparações, acredito que você já começou a pensar melhor sobre a engenharia de recursos e a perceber que devemos trabalhar com cada coluna dos nossos dados de forma exclusiva. Por exemplo, quando criarmos um algoritmo de *machine learning* para rodar em uma linha de produção, podemos utilizar um método de discretização para cada coluna.

Discretizando valores numéricos por frequência igual utilizando a biblioteca feature-engine

Diferente da biblioteca *scikit-learn*, em que utilizamos a mesma classe para criarmos a discretização por frequência igual, aqui utilizaremos a classe `EqualFrequencyDiscretiser`. Se você está utilizando o *Google Colaboratory*, terá que instalar a biblioteca *feature-engine* antes com o comando `!pip install feature-engine`.

Vamos importar a classe `EqualFrequencyDiscretiser` da biblioteca *feature-engine*:

```
from feature_engine.discretisation import EqualWidthDiscretiser
```

Vamos instanciar a classe `EqualFrequencyDiscretiser()` e, em seguida, ajustar o modelo aos nossos dados de treino.

```
discretizador_ft = EqualFrequencyDiscretiser(q=10,variables=
["Age","Fare"])
discretizador_ft.fit(X_treino)
```

Observe que, ao instanciarmos a nossa classe `EqualFrequencyDiscretiser`, ela recebeu dois parâmetros. O primeiro é referente ao número de *bins* ou intervalos que desejamos criar (utilizaremos 10 para criarmos o mesmo número de intervalos do exemplo anterior). O segundo parâmetro diz respeito às variáveis que desejamos discretizar, que, no nosso caso, são as variáveis `Age` e `Fare`.

Vamos inspecionar os intervalos calculados pelo nosso discretizador utilizando o comando `discretizador_ft.binner_dict_`:

```
discretizador_ft.binner_dict_
```

```
{'Age':[-
inf,18.0,21.0,25.0,27.0,28.0,32.0,39.80000000000001,48.900000000000034,inf
],
 'Fare':[-
```

```
inf,7.65,7.7958,8.05,10.5,13.8604,20.445000000000007,26.0,36.275000000000
3,69.09500000000016,inf]}
```

Por fim, vamos transformar nossos dados para podermos fazer nossas análises:

```
treino_ft = discretizador_ft.transform(X_treino)
teste_ft =discretizador_ft.transform(X_teste)
```

Os resultados dessas transformações podem ser vistos a seguir. Para os dados de treino:

```
treino_ft.head()
```

PassengerId	Pclass	Name	Sex	Age	SibSp
858	1	Peter Denis	male	51.0	0
53	1	Henry Sleeper	female	49.0	1
387	3	Sidney Leonard	male	1.0	5
125	1	Percival Wayland	male	54.0	0
579	3	Joseph	female	NaN	1

PassengerId	Parch	Ticket	Fare	Cabin	Embarke
858	0	113055	26.55	E17	S
53	0	PC 17572	76.7292	D33	C
387	2	CA 2144	46.9	NaN	S
125	1	35281	77.2875	D26	S

PassengerId	Parch	Ticket	Fare	Cabin	Embarke
579	0	2689	14.4583	NaN	C

E para os dados de teste:

```
teste_ft.head()
```

PassengerId	Pclass	Name	Sex	Age	SibSp
496	3	Gerious	male	NaN	0
649	3	Edward	male	NaN	0
279	3	Eric	male	7.0	4
32	1	William Augustus	female	NaN	1
256	3	Darwis	female	29.0	0

PassengerId	Parch	Ticket	Fare	Cabin	Embar
496	0	2627	14.4583	NaN	C
649	0	S.O./P.P. 751	7.55	NaN	S
279	1	382652	29.125	NaN	Q
32	0	PC 17569	146.5208	B78	C
256	2	2650	15.2458	NaN	C

Podemos, então, chamar a função `compara_proporcao` para compararmos as proporções do número de observações de cada intervalo.

```
compara_proporcao(treino_ft["Age"],teste_ft["Age"])
```

Sendo assim, as proporções para idade podem ser observadas na imagem a seguir:

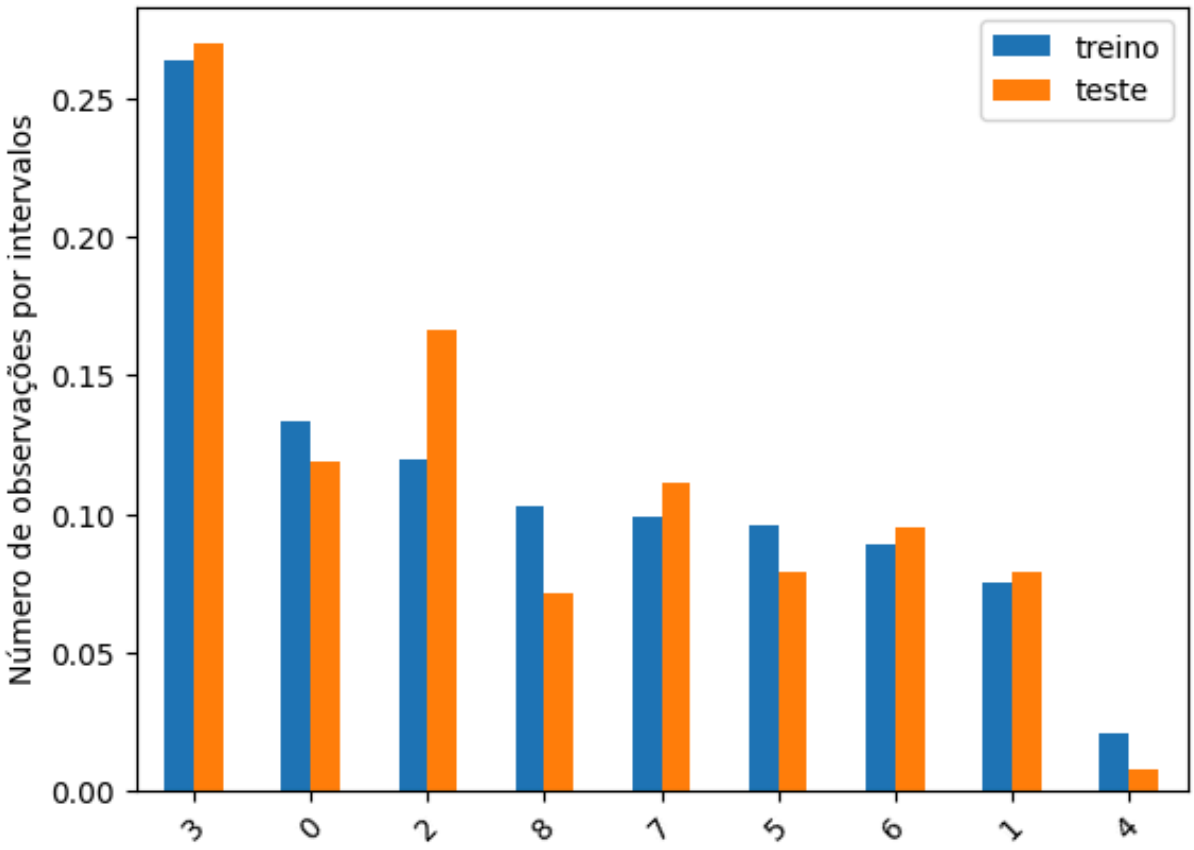

Figura 4.7: Resultado das proporções entre a variável Age.

Da mesma forma como na discretização com a biblioteca *scikit-learn*, temos alguns intervalos que possuem mais observações que outras, porém aqui está um pouco mais balanceado do que na discretização por valores iguais. Agora vamos chamar nossa função `compara_proporcao` novamente, desta vez para a coluna `Fare`:

```
compara_proporcao(treino_ft["Fare"],teste_ft["Fare"])
```

O resultado pode ser visto na imagem a seguir:

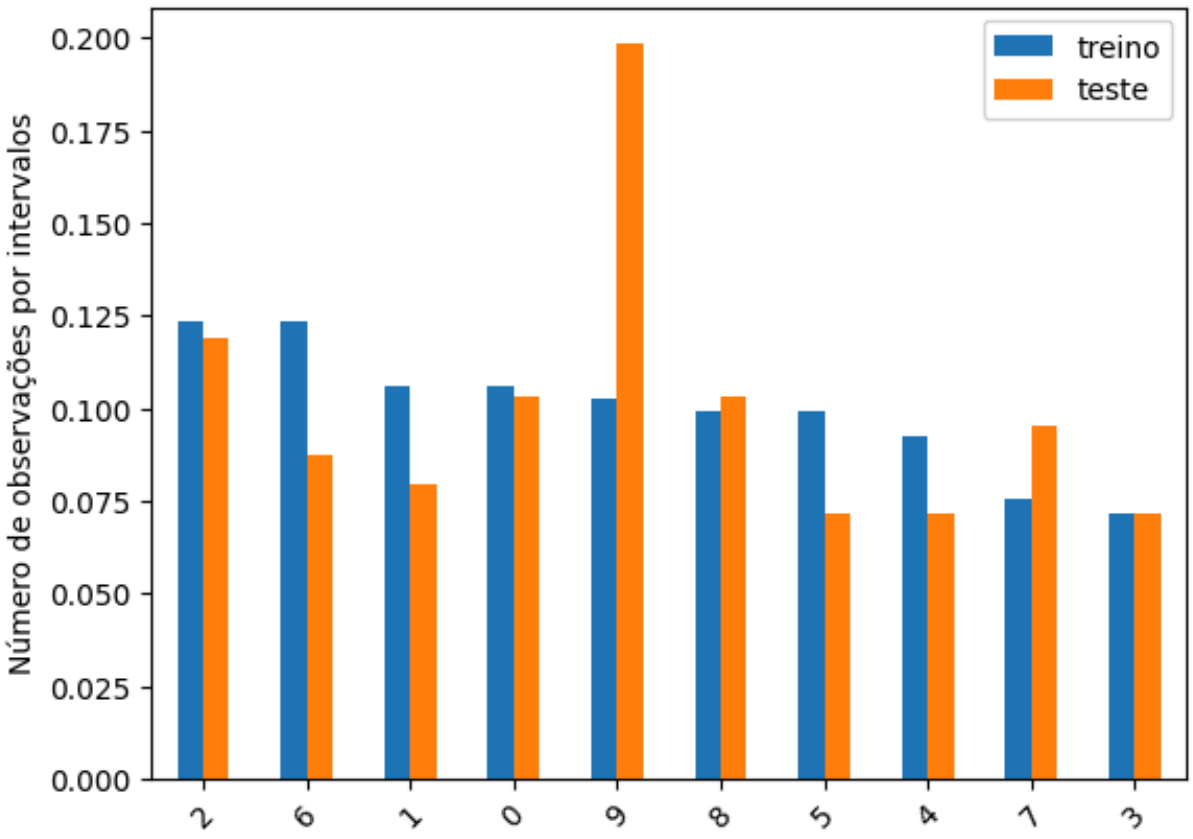

Figura 4.8: Porporção entre as observações da variável Fare.

Conseguimos observar que, com exceção do intervalo `9` na coluna de teste, todos os outros intervalos possuem uma proporção razoável em suas observações.

Discretização de valores numéricos utilizando k-means

Já vimos como utilizar a discretização por *k-means* com a biblioteca *scikit-learn*. Aqui veremos como efetuar a discretização em um projeto prático e como essa discretização afeta nossos conjuntos de treino e de teste.

Sendo assim, vamos dividir nosso conjunto de dados do Titanic em dados de treino e dados de teste utilizando o `train_test_split`, como no código a seguir:

```
from sklearn.model_selection import train_test_split

X_treino, X_teste, y_treino, y_teste =
train_test_split(dataframe.drop("Survived",axis=1),dataframe["Survived"],t
est_size=0.3,random_state=0)
```

O resultado dessa divisão nós já conhecemos. Vamos então fazer a importação da classe que utilizaremos para fazer a discretização:

```
from sklearn.preprocessing import KBinsDiscretizer
```

Vamos instanciar a classe importada:

```
discretizador =
KBinsDiscretizer(n_bins=4,encode="ordinal",strategy="kmeans")
```

Dentro do método `KBinsDiscretizer`, passamos os seguintes parâmetros:

- `n_bins` : utilizado para indicar o número de intervalos que desejamos criar;
- `encode="ordinal"` : utilizado para indicar o retorno dos intervalos em forma ordinal e não como um intervalo;
- `strategy` : indica a estratégia que desejamos utilizar na discretização.

Agora vamos ajustar o nosso modelo com o dataframe `X_treino` com as colunas que desejamos discretizar, ou seja, a coluna `Age` e a coluna `Fare`, como o código a seguir:

```
discretizador.fit(X_treino[["Age","Fare"]])
```

Vamos transformar nossos dados de treino e de teste e criar duas novas variáveis para armazenar os dados transformados:

```
treino_discretizado = discretizador.transform(X_treino[["Age","Fare"]])
teste_discretizado = discretizador.transform(X_teste[["Age","Fare"]])
```

Vale lembrar que a biblioteca *scikit-learn* devolve para nós um *numpy array*, sendo assim, se quisermos utilizar os métodos da

biblioteca *pandas*, devemos converter o *numpy array* para um *pandas dataframe*, da seguinte forma:

```
treino_discretizado = pd.DataFrame(treino_discretizado,columns=
["Age","Fare"])
teste_discretizado = pd.DataFrame(teste_discretizado,columns=
["Age","Fare"])
```

O resultado pode ser visto a seguir:

```
treino_discretizado.head()
```

Age	Fare
3.0	1.0
1.0	0.0
0.0	0.0
1.0	0.0
1.0	0.0

Para os dados de teste, temos:

```
teste_discretizado.head()
```

Age	Fare
0.0	1.0
1.0	0.0
3.0	0.0
1.0	0.0
2.0	0.0

Podemos inspecionar os valores que o `KBinsDiscretizer` ajustou utilizando o comando `discretizador.bin_edges_` :

```
discretizador.bin_edges_

array([array([0.17,19.25741499,34.13871001,50.41116942,76.]),
       array([0.,44.34356705,151.61194821,370.89877, 512.392])],dtype
=object)
```

Como nós discretizamos duas colunas, recebemos como retorno do `discretizador.bin_edges_` dois *arrays*, sendo o primeiro referente à coluna `Age` e o segundo, referente à coluna `Fare` . Podemos também inspecionar o número de intervalos criados utilizando o comando `discretizador.n_bins` :

```
discretizador.n_bins_

array([4,4])
```

Da mesma forma, o primeiro valor é referente à coluna `Age` e o segundo valor, à coluna `Fare` . Por fim, vamos criar os gráficos de comparação da proporção entre os intervalos e, para isso, vamos chamar a nossa função `compara_proporcao` . Primeiro, vamos comparar a proporção da variável `Age` . O resultado pode ser visto na figura a seguir:

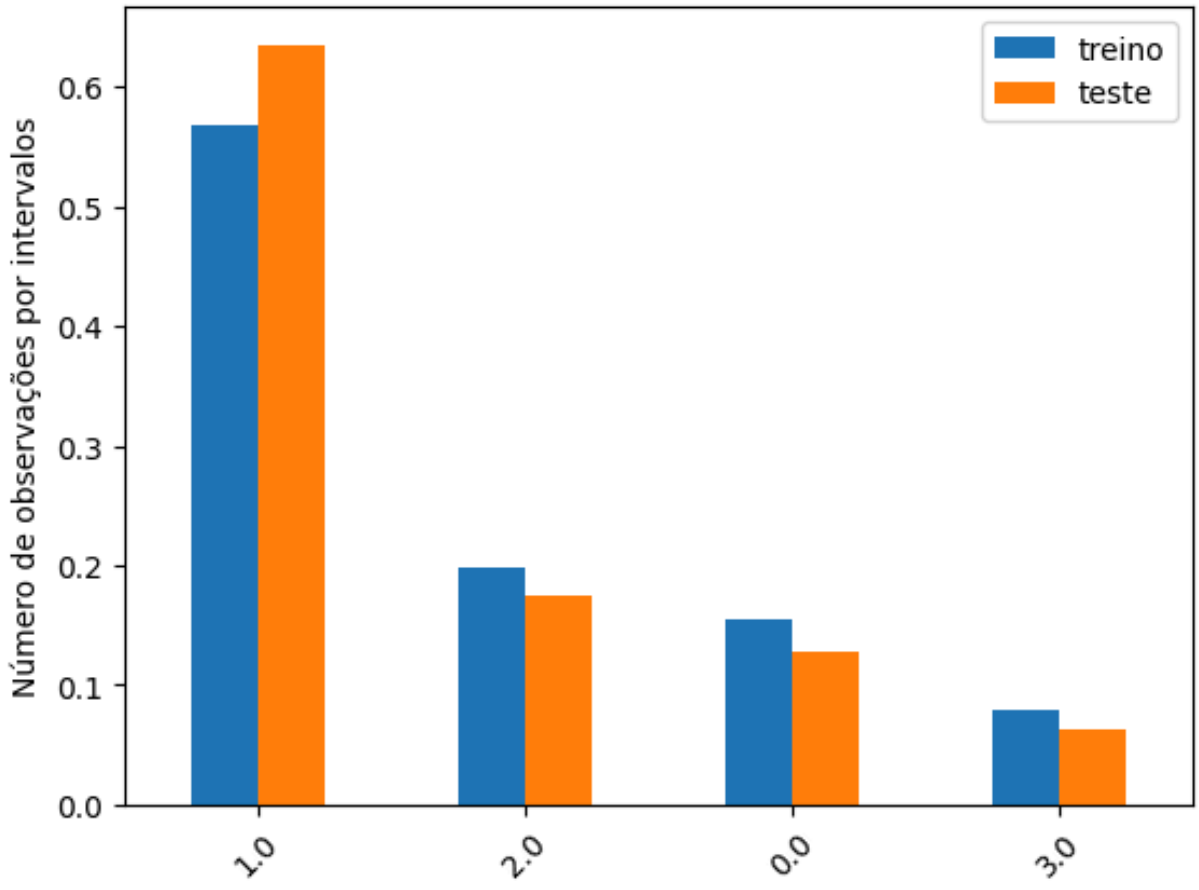

Figura 4.9: Comparação entre a proporção entre os intervalos criados.

Conseguimos observar que, nesse caso em específico, o número de observações dos dados de teste é maior no intervalo de número 1 e menor no intervalo de número 3. Além disso, conseguimos observar o desbalanceamento entre as nossas observações tanto nos dados de treino quanto nos dados de teste.

Vamos fazer essa mesma observação, porém agora com a variável `Fare`:

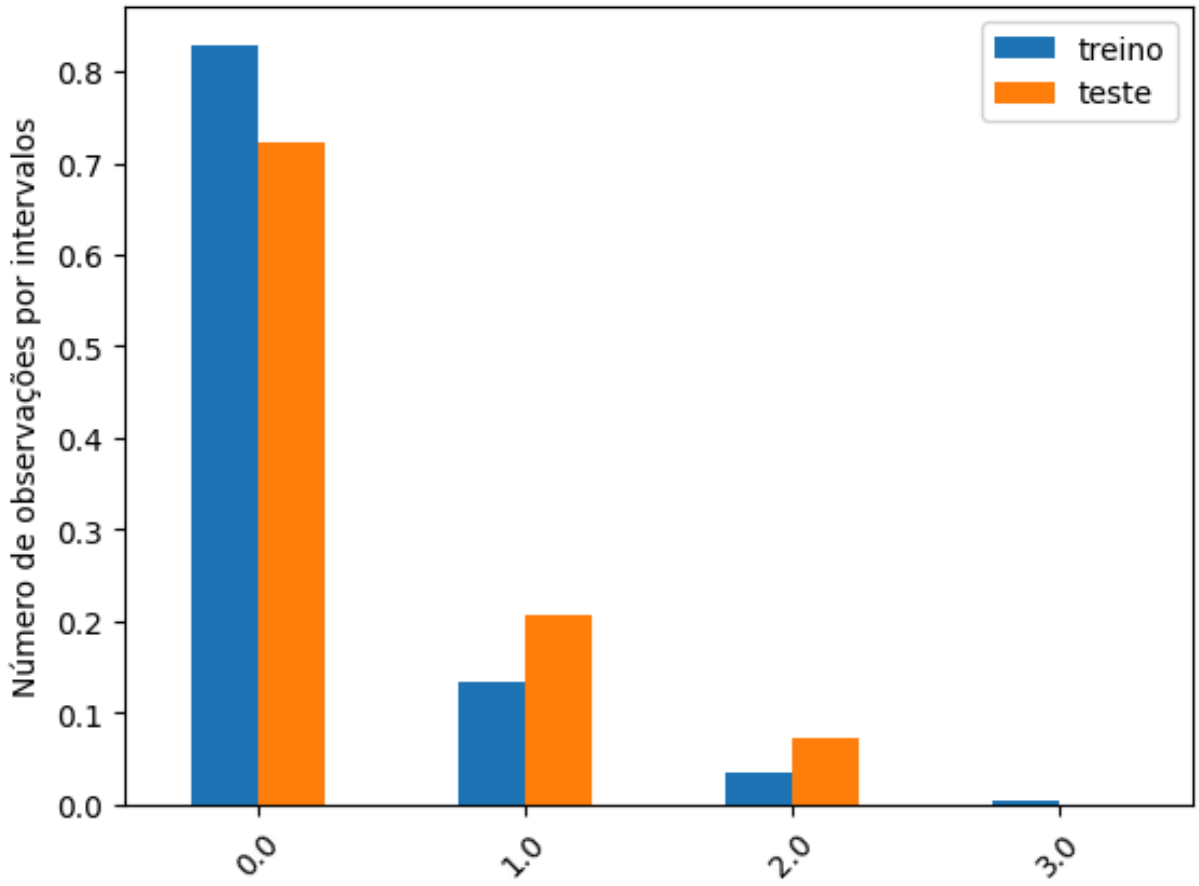

Figura 4.10: Comparação entre a proporção entre os intervalos criados.

No caso da coluna `Fare`, podemos observar que a proporção é pior do que a da coluna `Age`, porque temos um desbalanceamento maior entre as observações. Como dito anteriormente, a engenharia de recursos não é uma receita de bolo ou algo que vamos aplicar sempre da mesma forma. O que vai dizer o que aplicar e como aplicar são seus dados e a distribuição deles.

Na próxima seção, vamos ver como utilizar árvores para fazer a discretização das colunas de um dataframe.

4.4 Discretização por árvore utilizando a

biblioteca scikit-learn

Vamos aprender a utilizar a classe `DecisionTreeRegressor` da biblioteca *scikit-learn*. A discretização com árvores consiste em usar uma árvore de decisão para criarmos as caixas nas quais as nossas variáveis serão classificadas posteriormente.

A **árvore de decisão**, diferente dos outros processos que utilizamos até agora para fazer a discretização, utiliza a variável que queremos discretizar e a variável que desejamos prever para criar uma relação monótona entre elas.

Para este exemplo prático, utilizaremos um conjunto de dados diferente chamado `mercado.csv` , que pode ser encontrado no seguinte repositório do GitHub: https://github.com/ThiagoBenevides/feature-engine-transformando-dados-brutos-em-recursos/tree/main/capitulo_dois.

Nosso objetivo com a árvore de discretização é linearizar as variáveis, se possível. É importante ressaltar que nem sempre é possível transformar os dados em um conjunto perfeitamente linear; porém, podemos simplificá-los ao máximo para torná-los quase lineares, o que será altamente benéfico para nossos algoritmos de *machine learning*.

LINEAR E QUASE LINEAR

Dizemos que algo é *linear* quando existe uma relação direta e proporcional entre duas variáveis. Em outras palavras, quando uma variável aumenta, a outra também aumenta de maneira constante e previsível, seguindo uma linha reta no gráfico.

O termo *quase linear* refere-se a uma relação que se assemelha a uma linha reta, mas pode apresentar pequenas variações ou desvios em relação à linearidade perfeita. Nesses casos, podemos identificar uma tendência geral de aumento ou diminuição, mas pode haver dispersões ou imprecisões em nosso conjunto de dados.

Então, o primeiro passo para o nosso código é importar o nosso dataframe de estudo:

```
mercado = pd.read_csv("mercado.csv")
```

E o resultado desse código pode ser visto na tabela a seguir:

index	nome	idade	genero	renda	gasto
0	Gabriel Correia	42	Masculino	7507.22	363.97
1	Ana Sophia Cardoso	54	Feminino	7810.27	996.26
2	Augusto Castro	45	Masculino	5255.72	584.4
3	Enrico Alves	29	Masculino	8722.41	141.17
4	Mariana Silva	49	Feminino	5487.19	489.8

Antes de continuarmos, vamos entender o que cada coluna desse dataframe significa:

- `index` : representa o índice de nosso dataframe;
- `nome` : representa o nome do(a) cliente do mercado;
- `idade` : idade do(a) cliente;
- `genero` : gênero do(a) cliente. Obs.: para fins didáticos, este dataframe considera apenas gêneros binários;
- `renda` : renda mensal declarada pelo(a) cliente;
- `gasto` : gasto total no mês feito pelo(a) cliente do mercado.

Antes de criarmos o gráfico para analisarmos a relação entre a `renda` e o `gasto`, vamos normalizar os dados dessas colunas, pois, nos testes, o algoritmo de discretização se ajusta melhor aos dados normalizados. Então, vamos utilizar o algoritmo `StandardScaler` (veremos mais sobre este algoritmo na Parte 3, então não vamos entrar em muitos detalhes sobre ele agora).

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
colunas_para_normalizar = ['renda', 'gasto']
df_normalizado = mercado.copy()
df_normalizado[colunas_para_normalizar] =
scaler.fit_transform(mercado[colunas_para_normalizar])
```

O resultado deste processo é o novo dataframe a seguir:

nome	idade	genero	renda	gasto
Gabriel Correia	42	Masculino	0.81	-0.71
Ana Sophia Cardoso	54	Feminino	0.92	1.74
Augusto Castro	45	Masculino	-0.04	0.14
Enrico Alves	29	Masculino	1.27	-1.58
Mariana Silva	49	Feminino	0.040	-0.22

Com isso, podemos criar o gráfico da relação entre o `gasto` e a `renda`, como mostra o código a seguir:

```
plt.figure(figsize=(12,7))
df_normalizado.groupby("gasto")["renda"].mean().plot()
plt.title("Relação entre as variáveis gasto e renda")
plt.ylabel("Média dos gastos")
plt.xlabel("Renda")
```

O resultado dessa codificação pode ser visto na imagem a seguir:

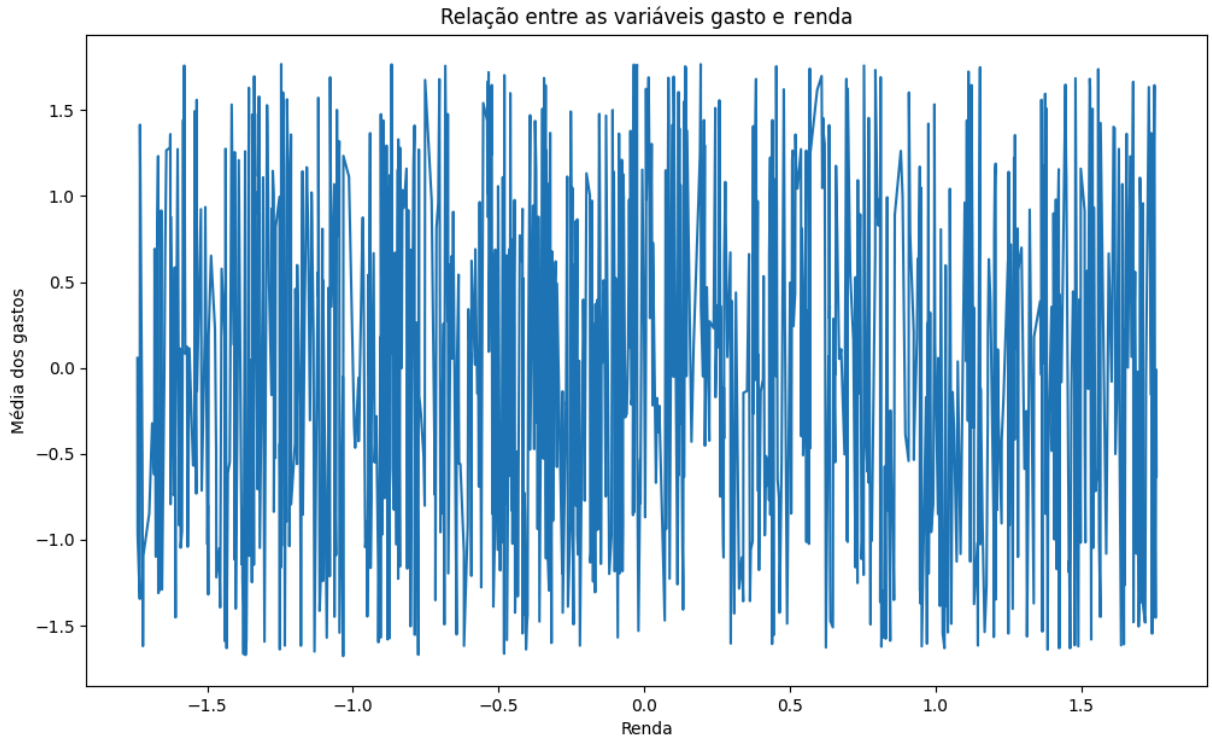

Figura 4.11: Relação entre as variáveis 'gasto' e 'renda'.

Observe que o gráfico não é uma linha reta, chegando até a ser um pouco caótico, no sentido de ser confuso de se interpretar. Então, precisamos fazer o nosso processo de linearização das variáveis `gasto` e `renda`; para isso, começaremos dividindo o nosso conjunto em dados de treino e dados de teste:

```
from sklearn.model_selection import train_test_split

X_treino, X_teste, y_treino, y_teste =
train_test_split(df_normalizado.drop("gasto",axis=1),df_normalizado["gasto
"],test_size=0.3,random_state=0)
```

Agora que dividimos o conjunto de dados em treino e teste, podemos começar o nosso processo de discretização utilizando árvore de decisão. Para isso, vamos importar da biblioteca `sklearn` o `DecisionTreeRegressor`:

```
from sklearn.tree import DecisionTreeRegressor
```

```
arvore = DecisionTreeRegressor(max_depth=2, random_state=0)
```

Com isso, conseguimos criar o nosso modelo de árvore com uma profundidade de 2. A **profundidade** da árvore diz respeito ao número de folhas que a nossa árvore vai ter.

Vamos ajustar o nosso modelo com a variável que desejamos discretizar e a variável-alvo, da seguinte forma:

```
arvore.fit(X_treino["renda"].to_frame(),y_treino)
```

Vamos transformar o nosso modelo e armazenar em uma nova variável, do seguinte modo:

```
renda_treino = arvore.predict(X_treino["renda"].to_frame())
renda_teste = arvore.predict(X_teste["renda"].to_frame())
```

Com isso, temos a variável `renda` discretizada e podemos construir novamente o gráfico para observarmos a relação entre a variável-alvo e a variável `gasto`. Podemos criar um dataframe com a discretização concluída com o código a seguir:

```
df_linear = pd.DataFrame({'renda': X_teste['renda'], 'gasto':
renda_teste})
```

Por fim, criaremos o gráfico com o dataframe `df_linear` para conseguirmos observar a nova relação entre as variáveis:

```
plt.figure(figsize=(12,7))
df_linear.groupby("gasto")["renda"].mean().plot()
plt.title("Relação entre as variáveis gasto e renda")
plt.ylabel("Média dos gastos")
plt.xlabel("Renda")
```

O resultado pode ser visto na imagem a seguir:

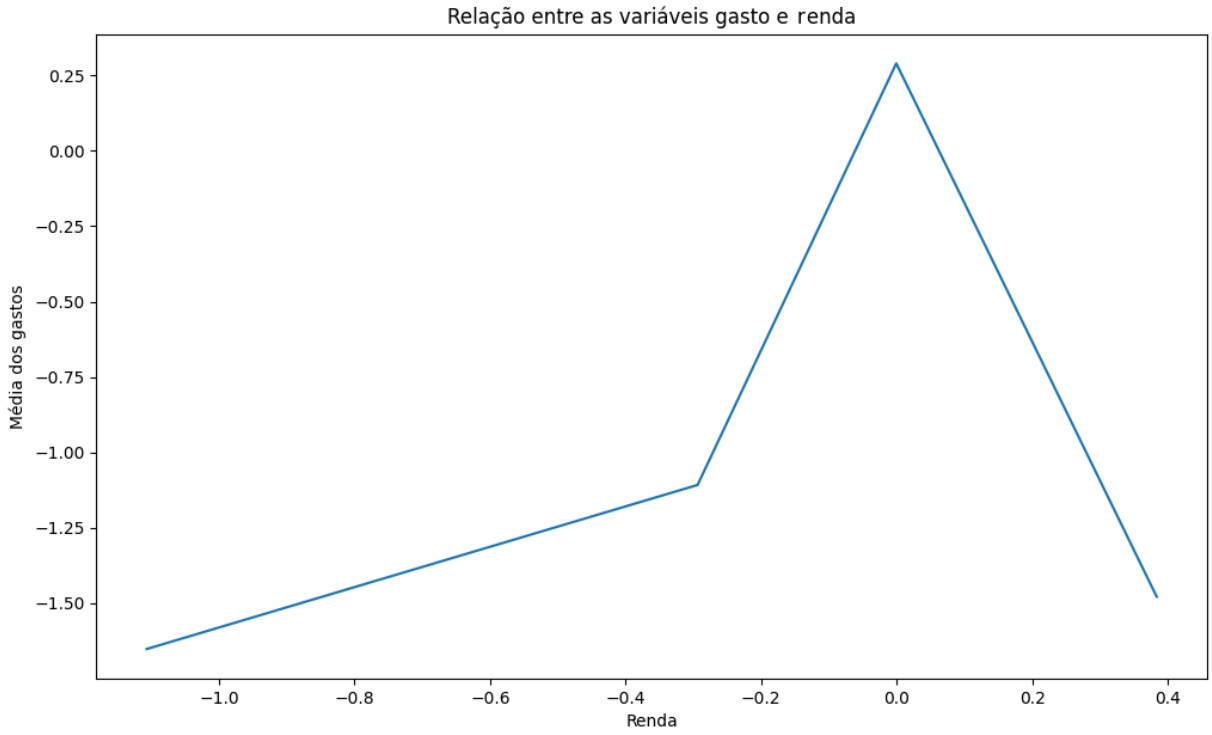

Figura 4.12: Relação entre as variáveis gasto e renda após a discretização.

Como podemos observar, o gráfico não é uma reta perfeita: ele ainda possui algumas quinas provocadas pela imperfeição dos dados. No entanto, está bem menos caótico e pode ser interpretado de uma forma muito mais simples. Na próxima parte, veremos como utilizar a biblioteca *feature-engine* para fazermos a discretização desse conjunto de dados.

4.5 Discretização via árvore com a biblioteca feature-engine

Vamos importar a classe que utilizaremos para fazer a discretização e instanciar a classe `DecisionTreeDiscretiser`, da seguinte forma:

```
from feature_engine.discretisation import DecisionTreeDiscretiser

arvore_ft = DecisionTreeDiscretiser(variables=['renda'], regression=True,
param_grid={'max_depth': [1, 2,3,4]})
```

Aqui, para instanciarmos a classe de `DecisionTreeDiscretiser`, passamos a única variável que desejamos discretizar, que nesse caso é a `renda`.

> **OBSERVAÇÃO**: o modo como escolhemos a profundidade máxima ainda é feito de forma aleatória. No último capítulo deste livro, veremos como você pode mensurar a precisão do algoritmo de *machine learning* e saber se esta foi uma boa escolha ou não.

Para a discretização do nosso conjunto de dados, iremos utilizar o método `fit` para ajustarmos a árvore de decisão e discretizar o conjunto de treino como mostrando no código a seguir:

```
arvore_ft.fit(X_treino, y_treino)
```

Nesse caso, passamos duas colunas, e o algoritmo vai interpretar os dados passados como um pandas dataframe; por esse motivo, não precisamos utilizar o método `to_frame()`.

Em seguida, usaremos o método `transform` para a discretização do nosso conjunto de dados. O método `transform` aplicará a discretização ao conjunto de teste, de forma consistente com o modelo treinado anteriormente com o método `fit`. Embora, o `predict` seja comum em outros contextos, como em previsões de valores, aqui estamos utilizando para transformar nossos dados em categorias discretas.

```
treino_disc = arvore_ft.transform(X_treino)
teste_disc = arvore_ft.predict(X_teste)
```

Com o código anterior criamos dois dataframes, chamados `treino_disc` e `teste_disc`, que armazenam os dados de treino e de teste já transformados. Podemos averiguar os parâmetros ajustados do nosso modelo com o comando `.binner_dict_`, e o resultado é o dicionário conforme mostrado a seguir:

```
treino_disc.binner_dict_
{'renda': GridSearchCV(cv=3, estimator=DecisionTreeRegressor(),
                       param_grid={'max_depth': [1, 2, 3, 4]},
                       scoring='neg_mean_squared_error')}
```

Analogamente, podemos fazer a mesma coisa com a variável `teste_disc` da seguinte forma: `teste_dict.binner_dict_`. Agora, vamos construir o gráfico para exibir a relação das variáveis `gasto` e `renda`:

```
df_disc = pd.DataFrame({'renda': treino_disc['renda'], 'gasto':
renda_treino})

plt.figure(figsize=(12,7))
df_disc.groupby("gasto")["renda"].mean().plot()
plt.title("Relação entre as variáveis gasto e renda")
plt.ylabel("Média dos gastos")
plt.xlabel("Renda")
```

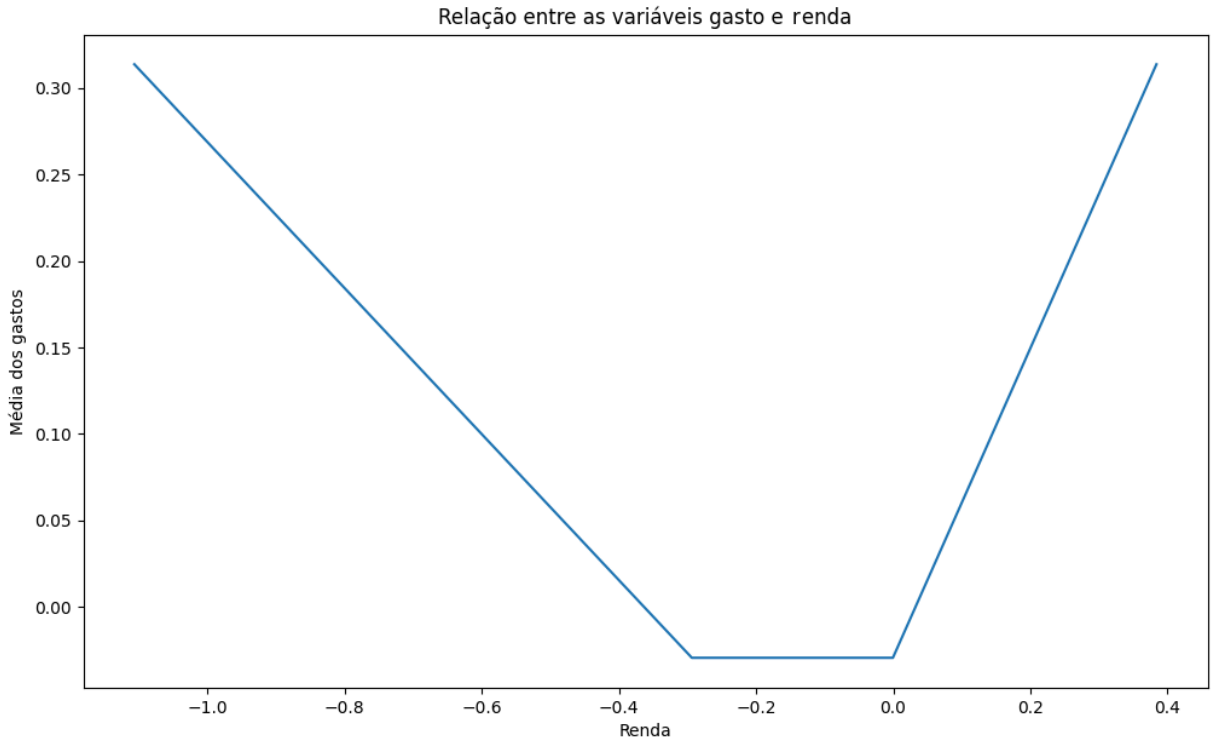

Figura 4.13: Relação entre as variáveis 'gasto' e 'renda'.

Com isso, conseguimos perceber que a biblioteca *feature-engine* não conseguiu transformar a relação entre as duas variáveis em uma reta perfeita, entretanto o modo como ela deixou a relação entre as duas variáveis já é uma grande ajuda aos nossos algoritmos de *machine learning* e também para nossas interpretações. Podemos interpretar esse gráfico da seguinte forma:

- Na primeira parte, que vai de `-1.1` até `-0.3`, a média dos gastos é decrescente;
- Na segunda parte, que vai de `-0.3` até `0.0`, a média dos gastos é constante;
- Na terceira parte, que vai de `0.0` até `0.4`, a média dos gastos é crescente.

Conseguimos ver que tanto o algoritmo da biblioteca *scikit-learn* quanto o da *feature-engine* cumprem muito bem o papel de linearizar a variável. No próximo capítulo, veremos como dimensionar recursos.

PARTE 3: Padronização de recursos

CAPÍTULO 5
Padronização de recursos

> Os materiais para este capítulo e para o projeto prático podem ser encontrados em: materiais da parte 3 (https://github.com/ThiagoBenevides/feature-engine-transformando-dados-brutos-em-recursos/tree/main/parte_tres)

Ao trabalharmos com certos conjuntos de dados podemos nos deparar com algumas dificuldades. Uma delas é a falta de padronização dos dados para que possamos compará-los de forma efetiva. Um dos fatores que provoca essa falta de padronização é a coleta de dados de várias fontes diferentes.

Para exemplificar isso, observe a tabela a seguir:

Date	FESA4.SA	BOVA11.SA	GRND3.SA
2020-01-02	17.28	114.24	8.83
2020-01-03	17.39	113.8	8.92
2020-01-06	17.13	112.59	8.81
2020-01-07	17.19	112.24	9.26
2020-01-08	17.37	111.95	9.24
2020-01-09	17.04	111.66	8.92
2020-01-10	17.34	111.28	8.9
2020-01-13	17.53	113.08	8.83
2020-01-14	17.81	113.32	8.93
2020-01-15	18.34	112.18	8.75

Essa tabela mostra os valores referentes aos preços das ações na Bolsa de Valores das empresas Bovespa, Fensa e Grendene em um dado período. Imagine que, com os valores dessa tabela, você tenha que escolher apenas uma dessas empresas para investir o seu dinheiro. Qual você escolheria e por quê? Olhando apenas para a tabela, pode ser um pouco difícil ou até mesmo inviável de se dizer, certo?

Você decide então fazer um gráfico para comparar os valores das ações ao longo do tempo, e o resultado que você tem é o seguinte:

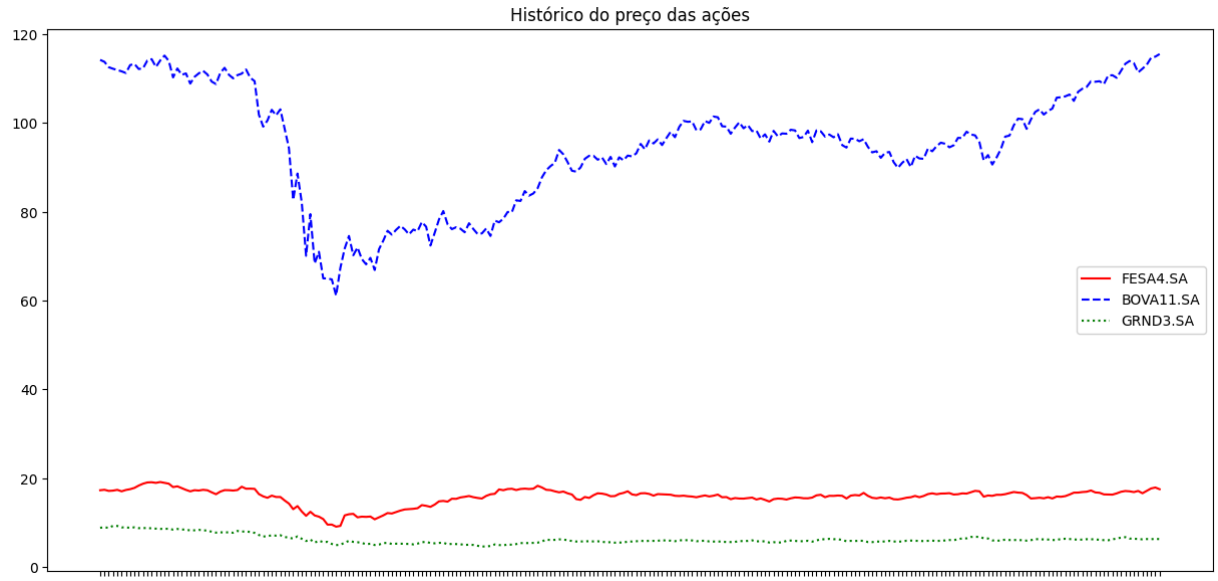

Figura 5.1: Gráfico com o valor das ações no decorrer do tempo.

Agora você consegue dizer em qual empresa você deve investir? Em um primeiro momento você pode ficar tentado a dizer que é a BOVA , pois, ao olhar para o gráfico, temos que o valor de suas ações é superior ao das outras. Porém, esse gráfico pode nos levar a uma falsa interpretação: a de que as ações da Bovespa possuem um melhor retorno.

A padronização serve para que os dados sejam colocados em uma mesma escala e assim tornar a comparação possível. Vamos a um exemplo. O que é maior: 1 légua ou 1 jarda? Pode ser difícil dizer dessa forma. Agora, se colocarmos todas essas medidas em uma mesma escala, a do metro, veja como fica a pergunta: o que é maior, 4828,03 metros (1 légua) ou 0,9144 metros (1 jarda)?

E na tabela anterior, qual padronização fizemos? Na tabela anterior, fizemos a padronização via normalização, que veremos a seguir. A normalização faz, grosso modo, a mesma coisa: ela transforma os valores para uma mesma escala, tornando a comparação possível.

Observe a imagem a seguir com o gráfico das mesmas empresas da tabela anterior, porém desta vez com normalização dos dados:

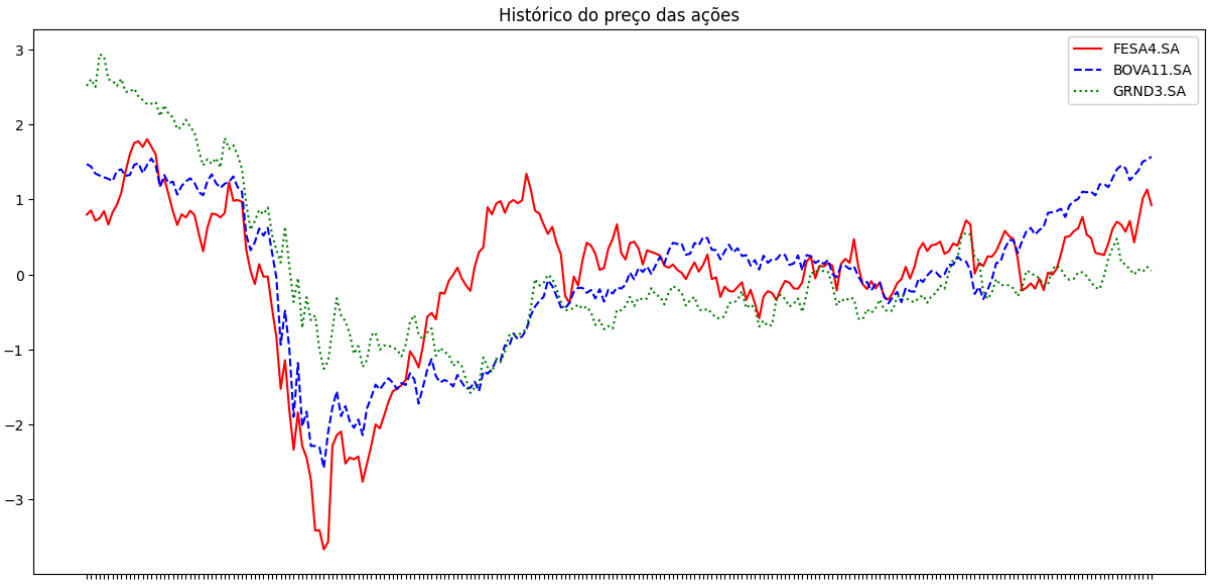

Figura 5.2: Ações dimensionadas para uma mesma escala.

A pergunta é: você ainda mantém a sua opção de investir na Bovespa? Essa imagem deixa claro o impacto de padronizarmos os dados. Porém, existe um outro fator que é ainda mais importante que o componente visual: alguns algoritmos de *machine learning* são sensíveis à magnitude dos números. Isto é, da mesma forma como você foi levado a acreditar que as ações da Bovespa eram superiores às outras duas, o algoritmo de *machine learning* pode ser induzido a erros. Por esse motivo, vamos aprender como trabalhar com a padronização dos dados neste capítulo.

5.1 Padronização pela normalização

Existem vários tipos de padronização de valores. O que vai dizer qual é o melhor são os dados e o desempenho do nosso algoritmo, como veremos no último capítulo, quando avaliarmos as métricas. Neste capítulo, vamos nos concentrar em apresentar alguns métodos de padronização, começando com o método da normalização.

A normalização dos dados consiste em ajustar os dados para uma distribuição normal com média 0 e desvio padrão 1. Podemos encontrar essa padronização na estatística como `z-score` também.

> **Distribuição normal**: na probabilidade ou estatística, a distribuição normal é uma das mais utilizadas para modelarmos fenômenos naturais.

Para fazermos a padronização por normalização, primeiro devemos encontrar a média do nosso conjunto de dados e, em seguida, o desvio padrão. Vamos fazer apenas para a coluna `BOVA11.SA` da tabela a seguir:

Date	FESA4.SA	BOVA11.SA	GRND3.SA
2020-01-02	17.28	114.24	8.83
2020-01-03	17.39	113.8	8.92
2020-01-06	17.13	112.59	8.81
2020-01-07	17.19	112.24	9.26
2020-01-08	17.37	111.95	9.24
2020-01-09	17.04	111.66	8.92
2020-01-10	17.34	111.28	8.9
2020-01-13	17.53	113.08	8.83
2020-01-14	17.81	113.32	8.93
2020-01-15	18.34	112.18	8.75

Vamos começar a nossa padronização calculando a média e o desvio padrão da `BOVA11.SA`. Para calcularmos a média da coluna em Python, podemos utilizar o método `np.mean()`, da seguinte forma:

```
import numpy as np

bova = [114.23,
113.80,112.58,112.23,111.94,111.66,111.27,113.08,113.32,112.18,112.34]
media_bova = np.mean(bova)
```

O resultado é `112.60`, então temos que a média da `BOVA11.SA` é de `R$ 112.60`, agora vamos calcular o desvio padrão e, para isso, a biblioteca `numpy` também conta com um método, que é o `np.std`. Ele é utilizado da seguinte forma:

```
desvio_padrao = np.std(bova)
```

Como resultado, temos que o desvio padrão da coluna `BOVA11.SA` é de `R$ 0.8694`, podendo ser arredondado para `R$ 0.87`. Por fim,

vamos fazer o processo de normalização em nossas colunas e, para isso, devemos seguir estes passos:

1. Calcular o desvio padrão da coluna que desejamos normalizar;
2. Subtrair do valor da linha pela média calculada;
3. Dividir o resultado da subtração pelo desvio padrão calculado;
4. Repetir o processo em todas as linhas.

Para fazermos esse processo vamos escrever o seguinte código:

```
bova_normalizada =[]
for i in range(len(bova)):
  normalizado = (bova[i]-media_bova)/desvio_padrao
  bova_normalizada.append(round(normalizado,2))
```

Com isso, temos a seguinte lista: `[1.87, 1.38, -0.03, -0.43, -0.76, -1.08, -1.53, 0.55, 0.82, -0.49, -0.3]`

Perceba que, depois de feita a normalização dos dados, podemos perder a referência dos valores, pois agora possuímos valores negativos em nossa coluna. Porém, mesmo sem essa referência, boa parte dos algoritmos de *machine learning* consegue ter um melhor desempenho, já que dão importância para a magnitude dos números — ou seja, quanto maior ele for, mais importante ele é para o algoritmo.

Agora que conseguimos entender o processo de padronização por normalização e o aplicamos em uma coluna, podemos aplicá-lo nas demais. Porém, em vez de repetir o código mais duas vezes, vamos criar uma função que recebe o dataframe com os valores que desejamos normalizar e, em seguida, aplicá-la em nossas colunas.

```
def normaliza(dataframe):
  dataframe_normalizado= dataframe.copy()
  colunas = dataframe.columns[1:]
  for coluna in colunas:
    media = round(np.mean(dataframe_normalizado[coluna]),2)
    std = round(np.std(dataframe_normalizado[coluna]),4)
    dataframe_normalizado[coluna] = (dataframe_normalizado[coluna]-
```

```
media)/std

  return dataframe_normalizado
```

Vamos criar um dataframe com os valores que desejamos normalizar:

```
data =["2020-01-02","2020-01-03","2020-01-06","2020-01-07","2020-01-
08","2020-01-09","2020-01-10","2020-01-13","2020-01-14","2020-01-
15","2020-01-16"]
bova = [114.23,
113.80,112.58,112.23,111.94,111.66,111.27,113.08,113.32,112.18,112.34]
fesa = [19.79,19.91,19.62,19.69,19.88,19.51,19.86,20.07,20.40,21.00,21.48]
grnd = [12.30,12.43,12.28,12.91,12.87,12.43,12.40,12.30,12.44,12.19,12.19]

dataframe =
pd.DataFrame({"data":data,"BOVA":bova,"FESA":fesa,"GRND":grnd})
```

Agora vamos testar nossa função `normaliza` no dataframe:

```
dataframe_normalizado = round(normaliza(dataframe),2)
```

Temos os seguintes valores que estão padronizados:

index	data	BOVA	FESA	GRND
0	2020-01-02	1.87	-0.55	-0.56
1	2020-01-03	1.38	-0.34	0.0
2	2020-01-06	-0.02	-0.83	-0.64
3	2020-01-07	-0.43	-0.72	2.06
4	2020-01-08	-0.76	-0.39	1.89
5	2020-01-09	-1.08	-1.02	0.0
6	2020-01-10	-1.53	-0.43	-0.13
7	2020-01-13	0.55	-0.07	-0.56
8	2020-01-14	0.83	0.49	0.04
9	2020-01-15	-0.48	1.52	-1.03

Vamos refazer esse processo de padronização dos dados, porém agora vamos construir uma função que recebe um dataframe e o padroniza:

```
data =["2020-01-02","2020-01-03","2020-01-06","2020-01-07","2020-01-
08","2020-01-09","2020-01-10","2020-01-13","2020-01-14","2020-01-
15","2020-01-16"]
bova = [114.23,
113.80,112.58,112.23,111.94,111.66,111.27,113.08,113.32,112.18,112.34]
fesa = [19.79,19.91,19.62,19.69,19.88,19.51,19.86,20.07,20.40,21.00,21.48]
grnd = [12.30,12.43,12.28,12.91,12.87,12.43,12.40,12.30,12.44,12.19,12.19]

dataframe =
pd.DataFrame({"data":data,"BOVA":bova,"FESA":fesa,"GRND":grnd})
dataframe.head()
```

O resultado da criação do dataframe pode ser visto na seguinte tabela:

data	BOVA	FESA	GRND
2020-01-02	114.23	19.79	12.30
2020-01-03	113.80	19.91	12.43
2020-01-06	112.58	19.62	12.28
2020-01-07	112.23	19.69	12.91
2020-01-08	111.94	19.88	12.87
2020-01-09	111.66	19.51	12.43
2020-01-10	111.27	19.86	12.40
2020-01-13	113.08	20.07	12.30
2020-01-14	113.32	20.40	12.44
2020-01-15	112.18	21.00	12.19
2020-01-16	112.34	21.48	12.19

Neste primeiro momento, faremos apenas a normalização da coluna `BOVA` para que você entenda o processo; após isso, faremos a normalização para todas as três colunas. O primeiro passo que faremos é uma cópia da nossa coluna, `bova`, da seguinte forma:

```
bova = dataframe["BOVA"].copy()
```

O resultado pode ser visto na seguinte lista:

```
[114.23,113.80,112.58,112.23,111.94,111.66,111.27,113.08,113.32,112.18,112.34]
```

Para padronizar a nossa coluna `bova`, vamos utilizar os métodos `np.mean` e `np.std` para calcularmos a média e o desvio padrão, respectivamente, como no código a seguir:

```
media = round(np.mean(bova),2)
std = round(np.std(bova),4)
```

Utilizamos o comando `round` para arredondarmos a média para `2` casas decimais e o desvio padrão para `4` casas decimais, para ficar similar ao nosso cálculo manual. Por fim, vamos padronizar a lista de valores `bova` da seguinte forma:

```
bova_normalizado = round((bova-media)/std,5)
```

Utilizamos novamente o método `round` para arredondarmos os valores padronizados para 5 casas decimais, e o resultado pode ser visto na lista:

```
[1.87486,1.38026,-0.02300,-0.42558,-0.75914,-1.08121,-1.52979,0.55210,0.82816,-0.48309,-0.029906]
```

Agora vamos calcular a média e o desvio padrão do `bova_normalizado`:

```
print(f"""Média: {round(np.mean(bova_normalizado),2)}
 Desvio padrão:{round(np.std(bova_normalizado),2)}""")
```

No código anterior você pode ter notado que utilizamos o comando `round` para exibir os valores da média e do desvio padrão arredondados com duas casas decimais. E o resultado pode ser visto a seguir:

```
Média: 0.0
Desvio padrão: 1.0
```

E aqui você deve reparar que o comando `round` foi utilizado para arredondar para `2` casas decimais. Isso acontece porque, ao longo dos cálculos que fizemos, acumulamos alguns erros devido ao arredondamento feito, e por conta desse acúmulo de erros da aritmética de ponto flutuante, os valores podem ser levemente alterados.

```
Média: -0.0
Desvio padrão: 1.0
```

Agora que vimos como padronizar uma coluna, vamos criar uma função chamada `normaliza` que faz essa padronização em todo o

nosso dataframe:

```
def normaliza(dataframe):
  dataframe_normalizado = dataframe.copy()
  colunas = dataframe.columns[1:]
  for coluna in colunas:
    media = round(np.mean(dataframe_normalizado[coluna]),2)
    std = round(np.std(dataframe_normalizado[coluna]),4)
    dataframe_normalizado[coluna] = (dataframe_normalizado[coluna]-
media)/std

  return dataframe_normalizado
```

O resultado da execução da função `dataframe_normalizado` pode ser visto na seguinte tabela:

data	BOVA	FESA	GRND
2020-01-02	1.87	-0.55	-0.56
2020-01-03	1.38	-0.34	0.0
2020-01-06	-0.02	-0.83	-0.64
2020-01-07	-0.43	-0.72	2.06
2020-01-08	-0.76	-0.39	1.89
2020-01-09	-1.08	-1.02	0.0
2020-01-10	-1.53	-0.43	-0.13
2020-01-13	0.55	-0.07	-0.56
2020-01-14	0.83	0.49	0.04
2020-01-15	-0.48	1.52	-1.03
2020-01-16	-0.3	2.33	-1.03

Vamos escrever uma função que calcula a média e o desvio padrão de todas as colunas do nosso dataframe normalizado:

```
def calcula(dataframe):
  colunas = dataframe.columns[1:]
  for coluna in colunas:
    print(f"""Ação: {coluna}
          Média: {round(np.mean(dataframe[coluna]),2)}
          Desvio padrão: {round(np.std(dataframe[coluna]),2)}""")
```

O resultado da execução dessa função pode ser visto no seguinte código:

```
calcula(dataframe_normalizado)

Ação: BOVA
          Média: 0.0
          Desvio padrão: 1.0
Ação: FESA
          Média: -0.0
          Desvio padrão: 1.0
Ação: GRND
          Média: 0.0
          Desvio padrão: 1.0
```

Assim, conseguimos inspecionar e ver que a média `0` e o desvio padrão `1` se mantêm em todo o nosso dataframe como queríamos. O próximo dimensionamento que veremos será o dimensionamento pela média.

5.2 Dimensionamento pela média

O dimensionamento pela média tem o objetivo de comprimir ou esticar o intervalo do nosso conjunto de dados para o intervalo de `[-1,1]` com centro igual a `0`. Para fazermos um dimensionamento pela média, devemos utilizar os seguintes passos:

1. Localizar os valores mínimo e máximo do nosso conjunto de dados;

2. Calcular a média e o desvio padrão do nosso conjunto.

Para conseguirmos localizar os valores mínimo e máximo da coluna `BOVA11.SA`, podemos utilizar os métodos `np.min` e `np.max` da biblioteca *numpy*, da seguinte forma:

```
minimo = np.min(bova)
maximo = np.max(bova)
media = dataframe["BOVA"].mean()
```

Com isso, obtemos os seguintes valores:

- Mínimo: 111.27
- Máximo: 114.23

Para realizarmos o dimensionamento pela média, devemos subtrair de cada um dos valores a média calculada e dividir o resultado pela subtração do valor máximo e valor mínimo, como mostra o código a seguir:

```
normalizados =[]
for i in range(len(bova)):
 aux = (bova[i]-media)/(maximo-minimo)
 normalizados.append(round(aux,2))
```

Como resultado, temos os seguintes valores, exibidos em forma de lista: `[0.55, 0.4, -0.01, -0.13, -0.22, -0.32, -0.45, 0.16, 0.24, -0.14, -0.09]`

Agora que entendemos como calcular o dimensionamento pela média, vamos fazer esse procedimento para todo o nosso conjunto de dados, ou seja, para as três colunas:

```
def dimen_media(dataframe):
  dataframe_normalizado= dataframe.copy()
  colunas = dataframe.columns[1:]
  for coluna in colunas:
    media = round(np.mean(dataframe_normalizado[coluna]),2)
    min = dataframe[coluna].min()
    max = dataframe[coluna].max()
```

```
    dataframe_normalizado[coluna] = round((dataframe_normalizado[coluna]-
media)/(max-min),2)

  return dataframe_normalizado

dataframe_norm = dimen_media(dataframe)
```

Utilizando a função que acabamos de criar em nosso dataframe, temos o seguinte resultado:

data	BOVA	FESA	GRND
2020-01-02	0.55	-0.16	-0.18
2020-01-03	0.41	-0.1	0.0
2020-01-06	-0.01	-0.25	-0.21
2020-01-07	-0.12	-0.21	0.67
2020-01-08	-0.22	-0.12	0.61
2020-01-09	-0.32	-0.3	0.0
2020-01-10	-0.45	-0.13	-0.04
2020-01-13	0.16	-0.02	-0.18
2020-01-14	0.24	0.15	0.01
2020-01-15	-0.14	0.45	-0.33
2020-01-16	-0.09	0.7	-0.33

Com isso, conseguimos concluir a padronização pela média e agora podemos criar novamente o gráfico para fazer as comparações entre as três ações e ver se houve alguma diferença significativa entre a padronização por normalização e a padronização pela média.

O gráfico desta normalização pode ser visto na imagem a seguir:

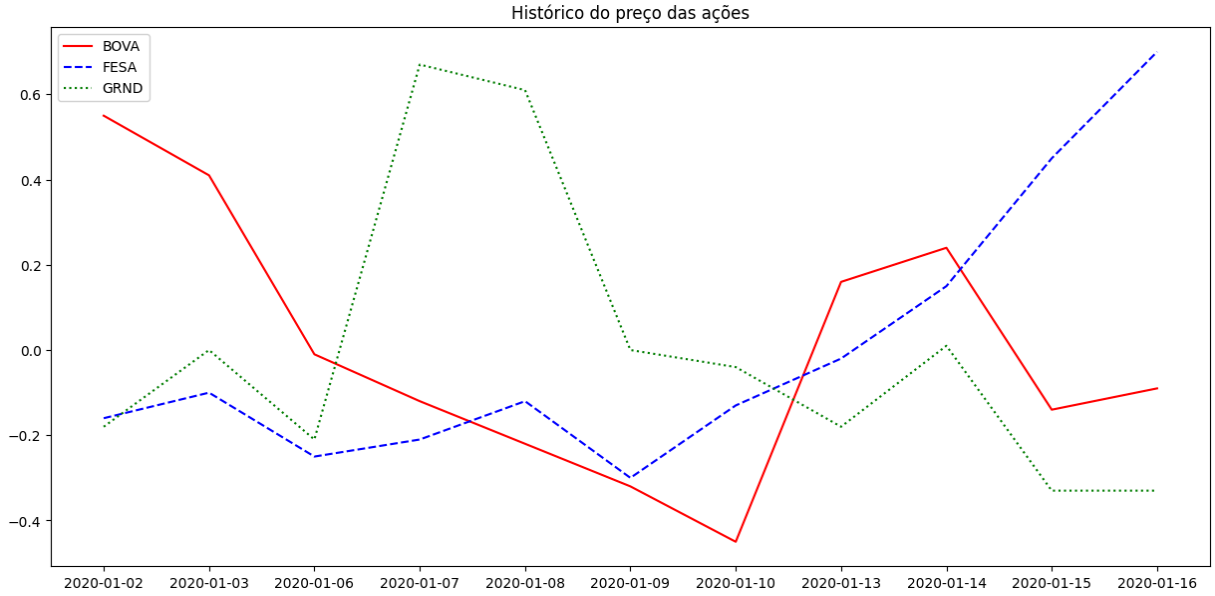

Figura 5.3: Gráfico com os valores padronizados pela média.

5.3 Padronização por min-max

A padronização que veremos nesta seção é chamada de padronização por min-max. É muito similar à anterior, porém aqui não utilizaremos a média, somente os valores mínimo e máximo. Da mesma forma como a padronização pela média, a padronização min-max também comprime ou estica o intervalo de números, porém ela faz isso no intervalo `[0,1]`.

Para fazermos essa padronização, subtraímos do valor da coluna o valor mínimo e dividimos esse resultado pela subtração entre o valor máximo e o mínimo, como segue a fórmula:

```
min_max = (x_i - min)/(max-min)
```

- `x_i` representa a *i-ésima* observação do nosso conjunto de dados;
- `min` representa o valor mínimo;
- `max` representa o valor máximo.

Vamos ao código em Python da padronização:

```
minimo= np.min(bova)
maximo = np.max(bova)
min_max = maximo - minimo
padronizados =[]
for i in range(len(bova)):
  aux = (bova[i]-minimo)/(maximo-minimo)
  padronizados.append(round(aux,2))
```

O resultado dessa padronização pode ser visto na seguinte lista:

```
[1.0, 0.85, 0.44, 0.32, 0.23, 0.13, 0.0, 0.61, 0.69, 0.31, 0.36]
```

Agora que conseguimos compreender a padronização utilizando a escala min-max, vamos fazer essa mesma padronização via código para as três colunas:

```
def min_max(dataframe):
  dataframe_normalizado= dataframe.copy()
  colunas = dataframe.columns[1:]
  for coluna in colunas:
    min = dataframe_normalizado[coluna].min()
    max = dataframe_normalizado[coluna].max()
    dataframe_normalizado[coluna] = round((dataframe_normalizado[coluna]-
min)/(max-min),2)

  return dataframe_normalizado
```

Executando o código da seguinte forma:

```
dataframe_min_max = min_max(dataframe)
```

O resultado pode ser visto na tabela a seguir:

data	BOVA	FESA	GRND
2020-01-02	1.0	0.14	0.15
2020-01-03	0.85	0.2	0.33
2020-01-06	0.44	0.06	0.12
2020-01-07	0.32	0.09	1.0
2020-01-08	0.23	0.19	0.94
2020-01-09	0.13	0.0	0.33
2020-01-10	0.0	0.18	0.29
2020-01-13	0.61	0.28	0.15
2020-01-14	0.69	0.45	0.35
2020-01-15	0.31	0.76	0.0

Uma ação pode valer 0.0?

Esse valor pode aparecer por dois motivos. O primeiro é o arredondamento que fizemos, que aqui deixamos com 2 casas depois da vírgula. O outro possível motivo é o modo como fazemos a conta, que nesse caso é subtrair o valor mínimo da nossa observação. Se nossa observação for o valor mínimo, teremos o mínimo menos o mínimo, que é igual a zero.

Por fim, podemos construir o nosso gráfico para compararmos às outras duas padronizações.

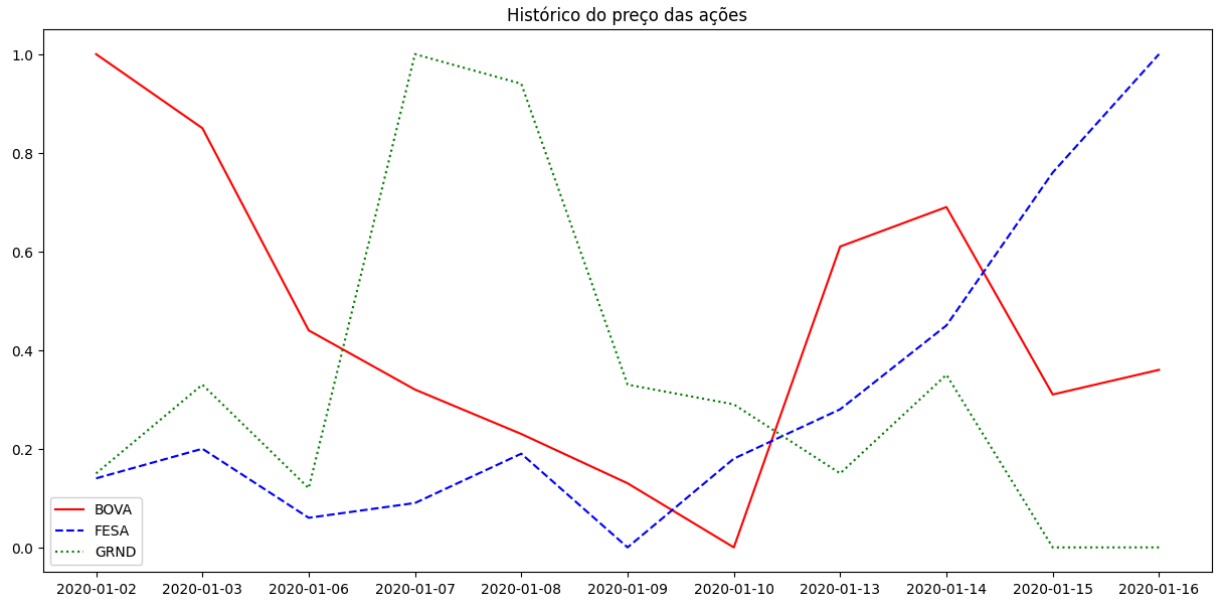

Figura 5.4: Gráfico com os dados padronizados.

Padronização utilizando a mediana e os quantis

A padronização que veremos nesta seção é também conhecida como **escala robusta**. Esse tipo de padronização é utilizado para conjuntos de dados que possuem muitos dados discrepantes, também chamados de *outliers*.

Esse método de padronização consiste em subtrair dos valores das linhas a mediana e dividir o resultado pela subtração do 75º quantil pelo 25º quantil. Como já foi dito, esse conceito de *quantil* vem da estatística. Os quantis são pontos estabelecidos em intervalos regulares que dividem os dados ordenados em `q` subconjuntos de dimensões essencialmente iguais. Não entraremos no mérito de calculá-los manualmente aqui.

Vamos ver isso na prática com o nosso conjunto de dados da `BOVA`. Vamos utilizar a nossa tabela de dados um pouco diferente do que temos utilizado para facilitar o entendimento. Observe a lista a seguir referente aos valores da coluna `BOVA`:

```
[114.23,113.80,112.58,112.23,111.94,111.66,111.27,113.08,113.32,112.18,112.34]
```

Vamos ordenar essa lista em ordem crescente:

`[111.27,111.66,111.94,112.18,112.23,112.34,112.58,113.08,113.32,113.80,114.23]`

Vamos encontrar a média do nosso conjunto de dados e então dividi-lo ao meio a partir da mediana. Dessa forma, teremos dois novos conjunto de dados, sendo o primeiro `[111.27,111.66,111.94,112.18,112.23]` e o segundo, `[112.58,113.08,113.32,113.8,114.23]`.

Vamos calcular a mediana desses dois conjuntos. A mediana do primeiro conjunto, e consequentemente o primeiro quantil, é o valor `111.94` e a mediana do segundo conjunto, consequentemente o terceiro quantil, é o valor `113.32`.

Agora podemos calcular o **intervalo interquartil**. Ele é uma medida estatística que serve para avaliarmos o quão disperso o nosso conjunto de dados está da mediana. Quanto maior for esse valor, mais dispersos esses valores estão, indicando a presença de muitos valores altos e baixos dentro do nosso conjunto de dados.

```
iqr = 113.32-111.94
iqr = 1.38
```

Vamos fazer a padronização pela mediana e o intervalo interquartil da primeira observação e apresentar a coluna `BOVA11` já padronizada. Para a primeira observação, temos:

```
escala = (111.27-112.30)(1.38)
escala = -(1.03)/(1.38)
escala = -0.74
```

Esta é a coluna `BOVA` já padronizada, aqui mostrada em forma de lista: `[1.39,1.86,0.20,-0.05,-0.26,-0.46,-0.74,0.56,0.73,-0.08,0.02]`

Para calcularmos os quantis, podemos utilizar um método da biblioteca `pandas` chamado `quantile`, que precisa de um único parâmetro, o quantil que desejamos capturar, como você pode ver no código a seguir:

```
quantil1 = dataframe["BOVA"].quantile(0.25)
quantil2 = dataframe["BOVA"].quantile(0.75)
```

Com isso, obtemos os seguintes valores:

- 25º quantil: 112.06
- 75º quantil: 113.19999999999999

Agora podemos fazer a normalização pelo quantil, da seguinte forma:

```
robusta = []
mediana_bova = np.median(bova)
for i in range(len(bova)):
  aux = (bova[i]-mediana_bova)/(quantil2-quantil1)
  robusta.append(round(aux,2))
```

O resultado pode ser visto na lista a seguir: `[1.66, 1.28, 0.21, -0.1, -0.35, -0.6, -0.94, 0.65, 0.86, -0.14, 0.0]`

Agora que vimos como fazer isso em uma coluna, vamos realizar essa padronização em nosso dataframe e, para isso, vamos criar a seguinte função:

```
def escala_robusta(dataframe):
  dataframe_copia= dataframe.copy()
  colunas = dataframe.columns[1:]
  for coluna in colunas:
    primeiro = dataframe_copia[coluna].quantile(0.25)
    terceiro = dataframe_copia[coluna].quantile(0.75)
    mediana = dataframe_copia[coluna].median()
    dataframe_copia[coluna] = round((dataframe_copia[coluna]-
mediana)/(terceiro-primeiro),2)

  return dataframe_copia
```

Podemos executar a função `escala_robusta` passando nosso dataframe para fazermos a padronização, como no código a seguir:

```
dataframe_quantile = escala_robusta(dataframe)
```

O resultado pode ser visto nesta tabela:

data	BOVA	FESA	GRND
2020-01-02	1.66	-0.18	-0.69
2020-01-03	1.28	0.06	0.21
2020-01-06	0.21	-0.53	-0.83
2020-01-07	-0.1	-0.38	3.52
2020-01-08	-0.35	0.0	3.24
2020-01-09	-0.6	-0.75	0.21
2020-01-10	-0.94	-0.04	0.0
2020-01-13	0.65	0.38	-0.69
2020-01-14	0.86	1.05	0.28
2020-01-15	-0.14	2.26	-1.45

Por fim, podemos criar o nosso gráfico comparativo:

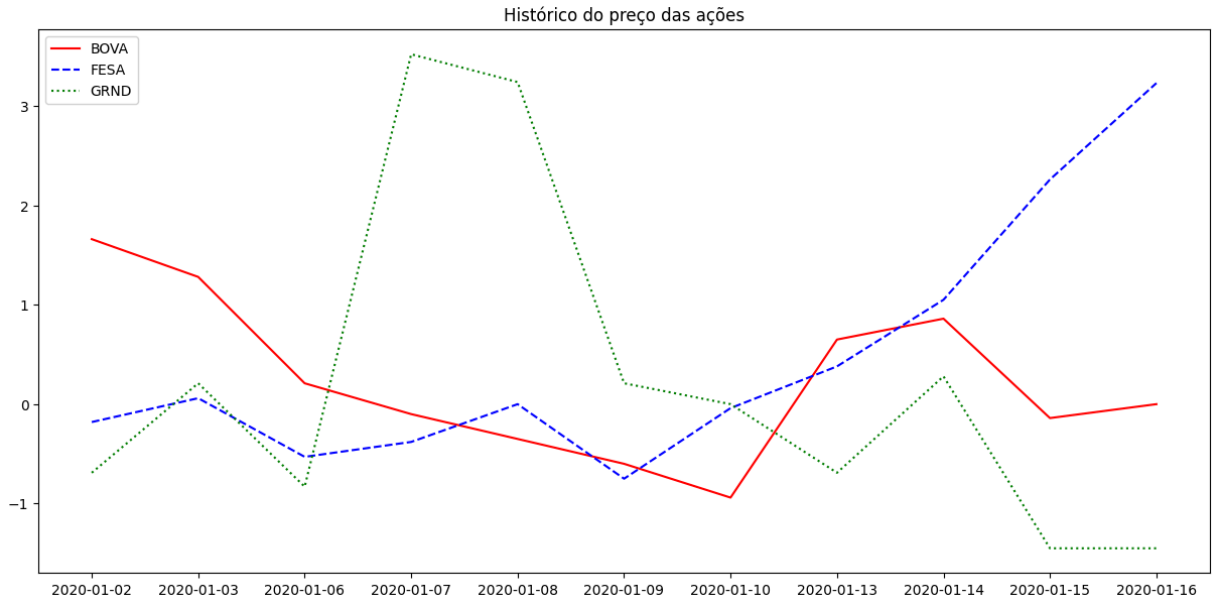

Figura 5.5: Gráfico comparativo utilizando a escala robusta.

Com isso, terminamos mais um capítulo. No próximo capítulo, faremos o nosso projeto prático e veremos uma breve introdução a como escolher a melhor padronização para alimentar o nosso modelo de *machine learning*.

Capítulo 6
Projeto prático — Padronização de valores de ações da Bolsa

Para o projeto prático desta parte, vamos coletar dados da Bolsa de Valores e padronizá-los utilizando a biblioteca *scikit-learn*. Porém para esse projeto precisaremos instalar a biblioteca *yfinance* para fazer a coleta dos dados com que vamos trabalhar.

Vamos começar instalando a biblioteca *yfinance*:

```
!pip install yfinance
import yfinance as yf
```

Após instalar e importar a biblioteca *yfinance*, vamos coletar os dados referentes às ações da Ferbasa como um teste para vermos como essa base de dados é importada. Para isso, vamos utilizar o código a seguir:

```
df_acoes = round(yf.download("FESA4.sa.SA",start="2020-01-01", end="2020-12-30"),2)
```

O código acima fez o download das ações da Ferbasa e o método `round` foi utilizado para arredondar os valores para duas casas decimais. Ao executarmos o código, teremos um dataframe como a tabela a seguir:

Date	Open	High	Low	Close	Adj Close	Volume
2020-01-02	36.99	36.99	36.47	36.85	36.85	42000
2020-01-03	36.85	37.25	36.66	37.15	37.15	18000

Date	Open	High	Low	Close	Adj Close	Volume
2020-01-06	36.95	37.96	36.95	37.96	37.96	43800
2020-01-07	37.94	38.22	37.79	38.08	38.08	108000
2020-01-08	38.0	38.24	38.0	38.20	38.20	46200

Antes de prosseguirmos com nosso projeto, vamos entender o significado de cada uma das colunas:

- `Open` : exibe o preço das ações no momento da abertura da Bolsa;
- `High` : exibe o maior preço atingido no dia;
- `Low` : exibe o menor preço atingido no dia;
- `Close` : exibe o valor do preço da ação no momento de encerramento da Bolsa;
- `Adj Close` : exibe os preços ajustados na hora do encerramento da Bolsa. O valor ajustado envolve a distribuição de dividendos;
- `Volume` : número de ações vendidas no dia.

Agora, vamos expandir nosso projeto e trabalhar com os dados de mais ações além dos dados da Ferbasa. Vamos incluir em nosso dataframe as seguintes empresas para efetuarmos o ajuste de dados do que seria uma carteira de ações.

- FESA4.SA: Ferbasa
- BOVA11.SA: Bovespa
- GRND3.SA: Grendene

Para padronizarmos os nossos dados, como já dito, vamos utilizar a biblioteca *scikit-learn*. Vamos começar importando os valores de

abertura das ações no caso a coluna `Open` das empresas com o código a seguir:

```
acoes = ["FESA4.SA","BOVA11.SA","GRND3.SA"]
carteira = pd.DataFrame()
for acao in acoes:
  carteira[acao] = round(yf.download(acao, start='2020-01-01',end="2020-12-31")['Open'],2)

carteira.to_csv("carteira.csv")
carteira =pd.read_csv("carteira.csv")
```

O código que acabamos de executar realiza duas tarefas:

- A primeira é o download dos valores de abertura das empresas do conjunto `acoes`, da data `2020-01-01` até a data `2020-12-31`;
- A segunda é armazenar com arredondamento de duas casas decimais em um arquivo `csv` e depois ler esse `csv`. Salvar em um arquivo `csv` garante que tenhamos esses valores salvos em um lugar local, não necessitando da API do Yahoo!.

Ao executarmos esse trecho de código, teremos um dataframe semelhante a este:

Date	FESA4.SA	BOVA11.SA	GRND3.SA
2020-01-02	17.28	114.24	8.83
2020-01-03	17.39	113.8	8.92
2020-01-06	17.13	112.59	8.81
2020-01-07	17.19	112.24	9.26
2020-01-08	17.37	111.95	9.24

A primeira coisa que faremos é escrever uma função para construirmos o gráfico com as ações. Faremos isso agora porque, ao longo desta parte prática, vamos construir vários gráficos.

Vamos ao código:

```
def grafico(carteira):
  linestyles = ['-', '--', ':']
  colors = ['red', 'blue', 'green']

  plt.figure(figsize=(15, 7))
  for i, col in enumerate(carteira.columns[1:]):
      if col != 'Date':
          plt.plot(carteira['Date'], carteira[col],
linestyle=linestyles[i], color=colors[i], label=col)


  plt.title('Histórico do preço das ações')
  plt.legend()


plt.show()

grafico(carteira)
```

O gráfico resultante pode ser visto na figura a seguir:

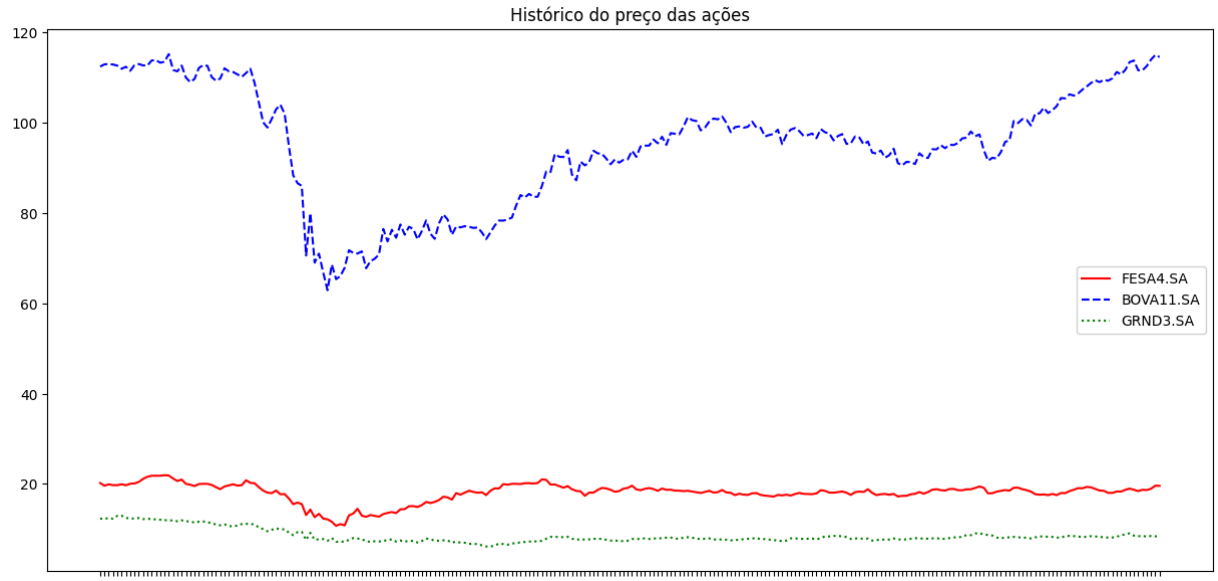

Figura 6.1: Gráfico com o preço das ações da carteira.

Agora que construímos nosso gráfico e analisamos seu comportamento, é hora de avançarmos no processo de análise. Vamos continuar com a chamada etapa de padronização dos nossos números. Isso é fundamental para garantir que os dados estejam em uma escala uniforme, o que é essencial para a aplicação de técnicas estatísticas e a realização de inferências em nosso conjunto de dados. Ao padronizá-los, estamos preparando o terreno para análises mais robustas e resultados mais significativos.

Nas próximas seções, exploraremos diferentes estratégias para a padronização dos dados e seus efeitos.

6.1 Padronização pela normalização utilizando a biblioteca scikit-learn

Vamos começar explicando a primeira forma de padronização, que é bastante simples e envolve a normalização. Essa técnica é conhecida como **Standard Scaler** e pode ser encontrada na biblioteca *scikit-*

learn. Vamos ver como essa padronização funciona. Para utilizá-la, vamos começar importando o Standard Scaler da biblioteca `sklearn`:

```
from sklearn.preprocessing import StandardScaler

padronizacao = StandardScaler()
```

Para esse método, não precisamos inserir nenhum parâmetro adicional, basta instanciarmos nossa classe `StandardScaler()`. Em seguida, vamos ajustar o nosso modelo aos dados que queremos transformar:

```
padronizacao.fit(carteira[carteira.columns[1:]])
```

Ajustamos os nossos dados com o comando `.fit` e passamos como parâmetro `carteira.columns[1:]`. Usamos esse comando para passar as colunas do nosso dataframe menos a coluna com os valores da data. Veja:

```
carteira.columns[1:]
Index(['FESA4.SA', 'BOVA11.SA', 'GRND3.SA'], dtype='object')
```

Para analisar as médias e os desvios padrões calculados pelo nosso `StandardScaler`, utilizamos os comandos `.mean_` e `.scale_`, da seguinte forma:

```
print(f"""Médias calculadas pelo StandardScaler: {padronizacao.mean_}
Desvios padrões calculados pelo StandardScaler: {padronizacao.scale_}
""")
```

O resultado desse código será similar a este:

```
Médias calculadas pelo StandardScaler: [18.03717742 95.05076613
8.47229839]
Desvios padrões calculados pelo StandardScaler: [ 2.06905914 12.97255816
1.52653966]
```

Agora podemos padronizar nossos dados com o código a seguir:

```
carteira_norm = pd.DataFrame()
carteira_norm = padronizacao.transform(carteira[carteira.columns[1:]])
carteira_norm = pd.DataFrame(carteira_norm, columns=carteira.columns[1:])
carteira_norm.insert(0, "Date", carteira["Date"])
```

No código anterior, fizemos o seguinte: primeiro criamos um dataframe vazio chamado `carteira_norm`. Em seguida, aplicamos a transformação aos dados da nossa `carteira` e os inserimos no dataframe `carteira_norm`. Alteramos os nomes das colunas do `carteira_norm` para que coincidissem com os nomes do dataframe original `carteira`. Por fim, inserimos a coluna `"Date"` na posição `0` do dataframe `carteira_norm`. Fizemos essas etapas para garantir compatibilidade com a nossa função `grafico`. O gráfico com os dados padronizados pode ser visto na imagem a seguir:

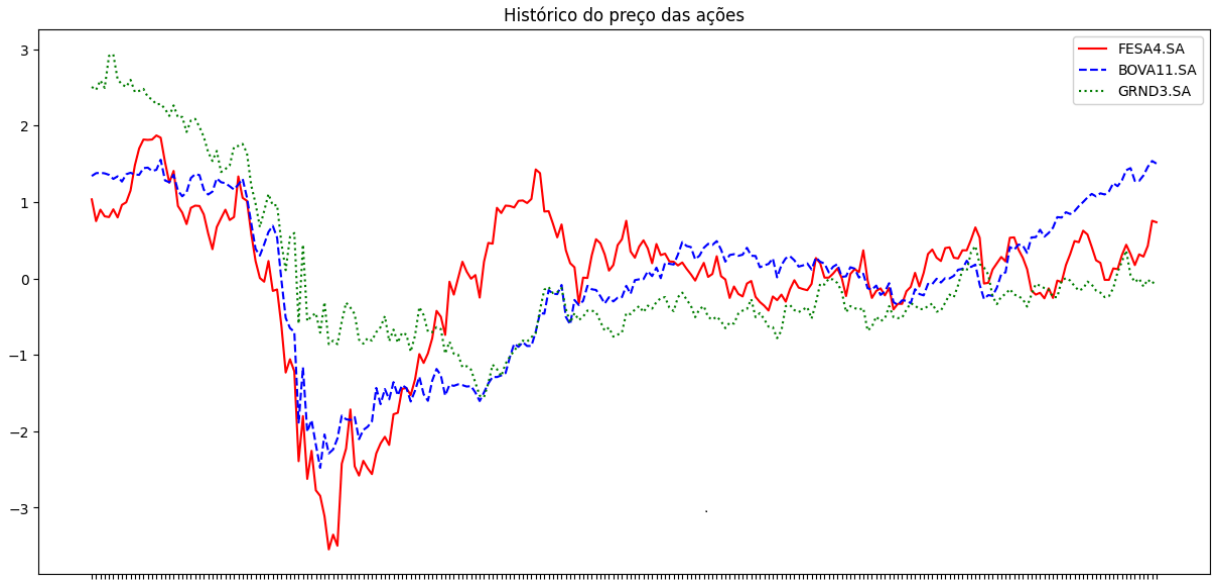

Figura 6.2: Gráfico com os preços das ações padronizados da carteira.

Veja que, ao padronizarmos os dados, nós alteramos a média e o desvio padrão deles. Terminada a padronização via normalização, devemos nos perguntar se esta é uma boa padronização. Para dizermos se ela é boa ou não, precisamos de alguma medida (ou *métrica*) para podermos compará-la com as demais. Para o nosso

exemplo prático, vamos utilizar a métrica conhecida como *Mean Absolute Error* (MAE), mas antes vamos criar um modelo de previsão para termos o que medir.

6.2 Criando um modelo de previsão ARIMA

Nosso objetivo aqui é criar um modelo de previsão utilizando o método de aprendizado de máquina chamado de **ARIMA**. O modelo ARIMA (*Autoregressive Integrated Moving Average*) é um modelo de previsão de séries temporais utilizado para analisar a evolução de uma variável ao longo do tempo e fazer previsões futuras com base em padrões históricos.

Utilizaremos esse modelo para prever o valor das ações da Bolsa e, assim, avaliar a precisão das diferentes técnicas de padronização que aprendemos ao longo deste capítulo. Desta forma, conseguiremos observar se existe alguma padronização que é melhor que a outra e como o algoritmo de *machine learning* reage a essas padronizações.

Para este exemplo, usaremos os mesmos dados que viemos utilizando:

Date	FESA4.SA	BOVA11.SA	GRND3.SA
2020-01-02	20.18	112.45	12.3
2020-01-03	19.59	112.93	12.24
2020-01-06	19.9	113.0	12.43
2020-01-07	19.72	112.9	12.28
2020-01-08	19.7	112.65	12.92

Vamos importar as bibliotecas necessárias para criarmos nosso algoritmo de aprendizado de máquina:

```
from statsmodels.tsa.arima.model import ARIMA
```

Agora vamos instanciar o nosso modelo ARIMA, porém neste momento não entraremos nos detalhes de construção do modelo:

```
modelo = ARIMA(carteira["BOVA11.SA"], order=(2, 0, 1))
```

Como dito, o modelo ARIMA é um modelo estatístico utilizado para analisar e prever séries temporais. Ele é composto por três componentes principais: Autorregressão (AR), Integração (I) e Média Móvel (MA).

- **Autorregressão (AR):** O componente AR utiliza valores passados da própria série temporal para prever valores futuros. O parâmetro `2` em `order` especifica a ordem da autorregressão, ou seja, quantos períodos anteriores são considerados para previsão. Nesse caso, são considerados os dois períodos anteriores.
- **Integração (I):** O componente I é usado para tornar a série temporal estacionária, ou seja, remover tendências e padrões não desejados. O parâmetro `0` indica que não há diferenciação aplicada à série temporal.
- **Média Móvel (MA):** O componente MA usa erros de previsão passados para prever valores futuros. O parâmetro `1` especifica

a ordem da média móvel, indicando quantos erros passados são levados em consideração para previsão.

Dessa forma, o modelo ARIMA criado com o código `ARIMA(carteira["BOVA11.SA"],order=(2,0,1))` utiliza os valores históricos da série temporal da ação `BOVA11.SA` para prever valores futuros, levando em conta a relação entre períodos anteriores (componente AR), sem a necessidade de diferenciação (componente I) e considerando erros de previsão passados (componente MA). Os parâmetros `(2, 0, 1)` especificam as ordens dos componentes AR, I e MA, respectivamente.

Vamos fazer as previsões para todo o dataframe:

```
resultados = modelo.fit()
previsoes=resultados.predict(start=-len(carteira))
```

Nesse código, fizemos três coisas:

- Ajustamos o modelo `ARIMA` aos dados da série temporal usando o método `fit()`. Ele estima os parâmetros do modelo com base nos dados fornecidos;
- Usamos o método `predict()` para fazermos a previsão. O parâmetro `start` define a posição inicial para a previsão. Nesse caso, estamos usando `-len(carteira)`, o que significa que a previsão será iniciada a partir do início dos dados da série temporal;
- Salvamos a previsão em uma variável `previsoes`.

Agora, vamos fazer as previsões para os próximos 30 dias com o seguinte código:

```
previsao = resultados.forecast(steps=30)
```

Nesse código, utilizamos o parâmetro `forecast` para fazermos a previsão e usamos o parâmetro `steps` com o valor `30` indicando que desejamos fazer a previsão para os próximos 30 dias das ações `BOVA11.SA`, que são as ações da Bovespa. O parâmetro `previsao` vem

em forma de lista; aqui, mostraremos os 5 primeiros resultados da previsão: `[114.578040,114.430685,114.364015,114.259782,114.174792]`.

Com isso, terminamos de fazer as previsões para os valores da Bovespa padronizados com a normalização. Podemos nos perguntar: este é um bom algoritmo de previsão? Ele consegue fazer boas previsões? Ainda não temos como saber se esse algoritmo, com essa padronização, é uma boa escolha. O que podemos fazer para isso é utilizarmos um método de avaliação para fazermos essa decisão. O método que utilizaremos nesse caso é chamado de *Mean Absolute Error*, ou, em uma tradução livre, *média do erro absoluto*.

Para utilizarmos essa métrica, precisaremos importá-la da biblioteca `sklearn`, da seguinte forma:

```
from sklearn.metrics import mean_absolute_error
```

E, para fazermos a avaliação, utilizaremos os valores reais, ou seja, os valores que coletamos do Yahoo! Finance com os valores previstos:

```
mean_absolute_error(carteira["BOVA11.SA"],previsoes)
```

Ao utilizarmos esse código, obteremos um valor de aproximadamente `1.43`. Isso significa que o nosso algoritmo está prevendo o valor da ação com uma diferença média de aproximadamente R$ 1,43 em relação ao valor real. Em outras palavras, em média, nossas previsões têm uma margem de erro de cerca de R$ 1,43 para mais ou para menos. Para visualizar essa diferença de forma mais simples, podemos dizer que estamos estimando o preço da ação com uma variação média de cerca de 60 centavos acima ou abaixo do valor real.

Na próxima seção, faremos a padronização *robusta* dos nossos dados.

6.3 Padronização com a mediana e quantis (escala robusta)

Nesta seção, vamos utilizar a classe `RobustScaler` da biblioteca *scikit-learn* para padronizar os valores do nosso dataframe `carteira`. Para começar, vamos importar a `RobustScaler` da biblioteca `sklearn`:

```
from sklearn.preprocessing import RobustScaler
```

Após a importação da nossa classe, podemos instanciá-la e ajustar o modelo passando como parâmetro o dataframe com as colunas que desejamos padronizar:

```
robusta = RobustScaler()
robusta.fit(carteira[carteira.columns[1:]])
```

Vamos inspecionar os valores calculados pelo método `RobustScaler` com os comandos `.center_` e `.scale_`, da seguinte forma:

```
print(f"""Centro calculado:{robusta.center_}
Escala calculada:{robusta.scale_}""")
```

O comando `.scale_` calcula o intervalo interquartil de cada uma das colunas e o comando `.center_` calcula o valor médio de cada uma das colunas passadas. O resultado pode ser visto a seguir:

```
Centro calculado:[18.4  96.63  7.93]
Escala calculada:[ 1.4675 14.925   0.85  ]
```

Agora podemos padronizar os dados com o código a seguir:

```
carteira_robusta = pd.DataFrame()
carteira_robusta = robusta.transform(carteira[carteira.columns[1:]])
carteira_robusta = pd.DataFrame(carteira_robusta,
columns=carteira.columns[1:])
carteira_robusta.insert(0, "Date", carteira["Date"])
```

Enfim, podemos gerar os gráficos com nossa função `grafico`. O resultado pode ser visto na imagem a seguir:

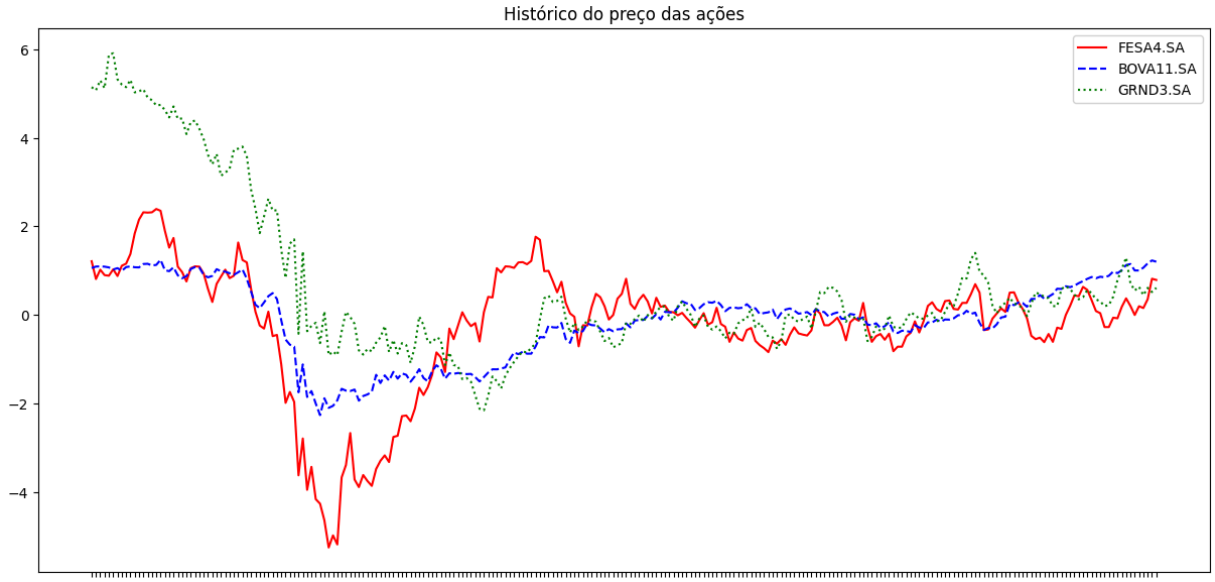

Figura 6.3: Gráfico com os preços das ações padronizados com a escala robusta da carteira.

Calculando o erro

Utilizaremos o modelo ARIMA para fazer as previsões e, com isso, conseguiremos comparar as duas padronizações. Vamos treinar novamente o nosso modelo, porém utilizando a escala robusta:

```
modelo_robusto = ARIMA(dataframe_robusto["BOVA11.SA"],order=(2,0,1))
results = modelo_robusto.fit()
```

Vamos fazer as previsões para os próximos 30 dias com o código a seguir:

```
previsoes_r =result.predict(start=-len(carteira_robusta))
previsao_r = result.forecast(steps=30)
```

O resultado que obtemos é uma lista de valores, mas aqui exibiremos apenas os 5 primeiros:

```
[1.202554,1.192686,1.188223,1.181244,1.175554] .
```

Por fim, vamos calcular o erro:

```
mean_absolute_error(dataframe_robusto["BOVA11.SA"],previsoes_r)
```

O erro que obtivemos com essa padronização foi de `0.09565693819441047`, ou seja, nosso modelo treinado com a escala robusta tem o erro de R$ 0,09 para mais ou para menos. Então, podemos afirmar que, neste caso em particular, a padronização robusta teve resultados melhores que a padronização por normalização.

Neste tópico, vimos de forma resumida como podemos avaliar as padronizações e saber qual delas é a melhor. Claro que podemos melhorar ainda mais esses resultados, porém vamos conversar sobre essa parte mais à frente, quando discutirmos as métricas de avaliação.

6.4 Padronização com a média

Infelizmente, a biblioteca *scikit-learn* não conta com nenhuma classe para executarmos diretamente a padronização pela média; no entanto, podemos fazer essa padronização de forma indireta. Para isso, utilizaremos o `StandardScaler` e o `RobustScaler` mudando alguns parâmetros desses dois métodos.

Vamos começar com as importações:

```
from sklearn.preprocessing import StandardScaler, RobustScaler
```

Vamos instanciar a nossa classe `StandardScaler`:

```
standardscaler = StandardScaler(with_mean=True, with_std=False)
```

O parâmetro `with_mean` faz com que o método diminua a média das observações. O parâmetro `with_std` como `False` faz com que o resultado da subtração não seja dividido pelo desvio padrão.

Vamos instanciar a nossa classe `RobustScaler`:

```
robustscaler = RobustScaler(with_centering=False,with_scaling=True,
quantile_range=(0,100))
```

Quando declaramos o parâmetro `with_centering` como `False`, estamos solicitando ao `RobustScaler` que ele não calcule o centro da nossa padronização. O parâmetro `with_scaling` faz com que seja calculada a escala robusta do nosso conjunto de dados e, ao declararmos `(0,100)` no parâmetro `quantile_range`, fazemos com que todo o nosso conjunto de dados seja utilizado no cálculo em vez do intervalo interquartil.

Agora que instanciamos as nossas classes, podemos treiná-las utilizando o nosso dataframe:

```
standardscaler.fit(carteira[carteira.columns[1:]])
robustscaler.fit(carteira[carteira.columns[1:]])
```

Com os nossos dados treinados, podemos transformá-los, lembrando que desta vez não faz sentido utilizarmos o comando `robustscaler.center_`, já que marcamos `with_centering` como `False`. Então, vamos transformar nosso conjunto de dados da seguinte forma:

```
dataframe_media =
standardscaler.transform(robustscaler.transform(carteira[['FESA4.SA',
'BOVA11.SA', 'GRND3.SA']]))
```

O código que acabamos de executar faz com que primeiro seja executada a padronização robusta e o resultado seja padronizado novamente, porém com a padronização pela normalização. Vamos converter esses dados em um dataframe e, em seguida, criar os gráficos novamente:

```
carteira_media = pd.DataFrame()
carteira_media = robusta.transform(carteira[carteira.columns[1:]])
carteira_media = pd.DataFrame(dataframe_media,
columns=carteira.columns[1:])
carteira_media.insert(0, "Date", carteira["Date"])
```

O resultado desse código é algo similar à seguinte tabela:

Date	FESA4.SA	BOVA11.SA	GRND3.SA
2020-01-02	-16.23	-92.90	-6.67
2020-01-03	-16.28	-92.89	-6.68
2020-01-06	-16.26	-92.89	-6.65
2020-01-07	-16.27	-92.89	-6.67
2020-01-08	-16.27	-92.89	-6.58

VALORES NEGATIVOS?

Quando aplicamos uma escala robusta, é comum obter valores negativos nos dados transformados. Isso ocorre devido ao processo de centralização e escala realizados pela escala robusta. Então, o que obtivemos em nosso dataframe `carteira_media` não é um erro.

Antes de calcularmos o erro, vamos construir o nosso gráfico com a função `grafico`. O resultado pode ser visto na imagem a seguir:

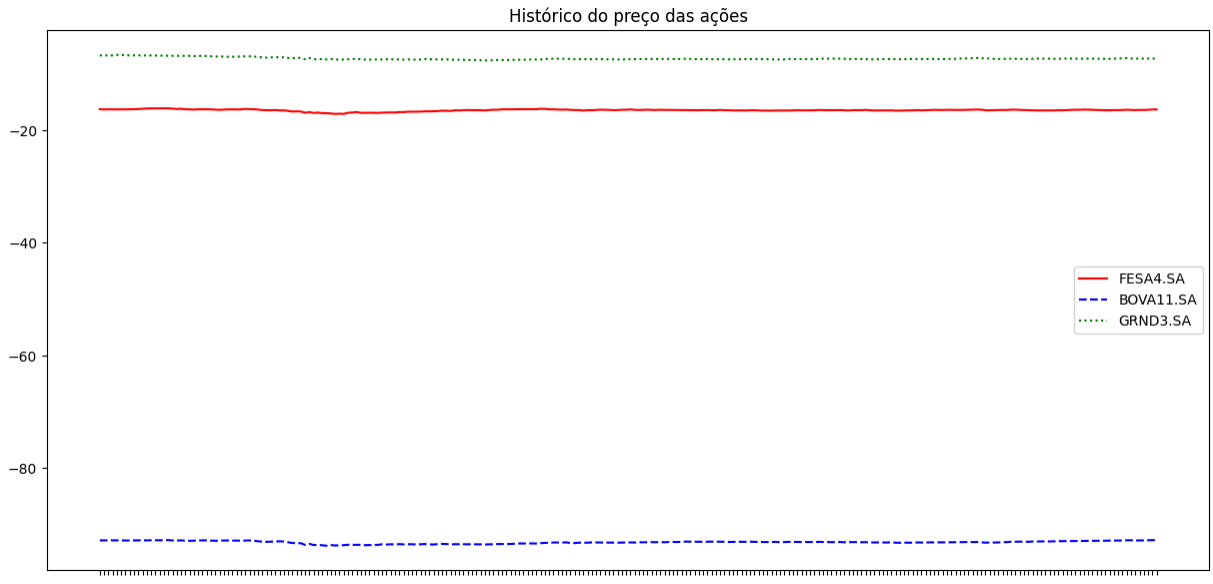

Figura 6.4: Gráfico com o preço das ações padronizados com a escala robusta da carteira.

GRÁFICO QUASE COMO UMA LINHA?

O gráfico que acabamos de fazer aparece quase como uma linha reta. Isso pode ser devido ao fato de que o modelo que nós criamos não consegue capturar as variações ou padrões presentes nos dados. É possível que o modelo criado esteja sofrendo de um problema de subajuste (*underfitting*), o que significa que a complexidade do modelo não é suficiente para capturar a variabilidade dos dados. Algumas coisas que podemos fazer para arrumar o modelo são:

- Considerar o ajuste dos parâmetros do modelo ARIMA;
- Avaliar se a quantidade de dados disponível é suficiente para treinar e validar o modelo.

Agora, sim, vamos calcular o erro.

Calculando o erro

Por fim, vamos calcular o erro com a média. Desta forma, conseguiremos comparar as três padronizações. Vamos começar criando o modelo ARIMA para a padronização feita pela média e, em seguida, ajustar os nossos dados ao modelo criado:

```
modelo_media = ARIMA(dataframe_media["BOVA11.SA"],order=(2,0,1))
resultado_media = modelo_media.fit()
```

Vamos criar as nossas previsões:

```
previsoes_m =resultado_media.predict(start=-len(carteira_media))
previsao_m = resultado_media.forecast(steps=30)
```

Os 5 valores previstos podem ser observados pela seguinte lista: `[-92.861245,-92.864911,-92.866917,-92.869719,-92.872069]`. Para conferirmos se essas previsões foram boas previsões, vamos calcular o erro cometido com o seguinte código:

```
mean_absolute_error(carteira_media["BOVA11.SA"],previsoes_m)
```

O erro cometido por essa padronização foi de `0.027349397672241614`, ou seja, obtivemos um erro de R$ 0.02 centavos para mais ou para menos nos valores das ações da Bovespa. A tabela a seguir faz a comparação dos 3 erros cometidos pelas padronizações que fizemos em relação as ações da Bovespa:

Normalização	**Robusta**	**Média**
1.42	0.09	0.02

Com essas métricas em mãos, podemos concluir que a melhor padronização para o nosso conjunto de dados foi a padronização pela média, já que foi a que teve o menor erro. Porém, não se engane: nem sempre a padronização pela média será a melhor padronização. O que vai dizer isso é o nosso conjunto de dados e os testes feitos. Mais à frente, quando discutirmos como escolher a melhor padronização para inserir em algoritmos de *machine learning*, retornaremos a este exemplo e então veremos, de forma prática, como a padronização pode interferir ou não em nossas predições.

PARTE 4: Codificando valores categóricos

CAPÍTULO 7
Codificação de valores categóricos

> O notebook e os materiais deste capítulo podem ser encontrados em materiais da parte 4 (https://github.com/ThiagoBenevides/feature-engine-transformando-dados-brutos-em-recursos/tree/main/parte_quatro)

Algoritmos de *machine learning* dificilmente consomem valores diferentes de valores numéricos. Porém, com o crescimento das redes sociais, dados não numéricos como fotos, vídeos, *likes*, entre outros, são compartilhados de forma mais intensa. Neste capítulo, veremos como podemos trabalhar com esses valores que não são consumidos por algoritmos de *machine learning*.

7.1 Codificação One-Hot

A codificação One-Hot que veremos nesta seção transforma os recursos do tipo categórico com n valores em n recursos, de tal forma que apenas um deles seja verdadeiro em um determinado momento.

Para que você compreenda melhor como é feita a codificação One-Hot, vamos fazer a transformação em um conjunto de dados hipotéticos de um curso EAD com aulas síncronas, ou seja, as aulas

são ministradas em uma hora em específico. A tabela que utilizaremos é mostrada a seguir:

Nome	Turno
Paula	Primeiro
Ana	Segundo
Denise	Primeiro
Arthur	Segundo
Maria	Primeiro
Thiago	Segundo
Alan	Primeiro
Catarina	Primeiro
Marcos	Segundo
José	Primeiro

O método One-Hot transforma `n` categorias distintas em `n` colunas. Por exemplo, se fizermos a codificação One-Hot na coluna `Turno`, teremos o seguinte resultado:

```
|Nome|Primeiro|Segundo|
```

Como temos duas categorias, `Primeiro` e `Segundo`, tivemos como resultado duas colunas, uma para cada uma das categorias. Porém, com quais valores essas colunas serão preenchidas? Onde estiver `Primeiro` na coluna `Turno` da tabela original, na coluna `Primeiro` da tabela One-Hot será preenchido com o valor `1` e a coluna `Segundo` será preenchida com `0`. No caso de o registro na coluna `Turno` estar preenchido com `Segundo` na tabela original, então a coluna `Segundo` da tabela One-Hot será preenchida com `1` e a coluna `Primeiro` será preenchida com `0`, conforme vemos a seguir:

Nome	Primeiro	Segundo
Paula	1	0
Ana	0	1
Denise	1	0
Arthur	0	1
Maria	1	0
Thiago	0	1
Alan	1	0
Catarina	1	0
Marcos	0	1
José	1	0

Prosseguindo, vamos partir para a codificação *One-Hot* em Python para analisar como ela funciona. Começaremos criando o nosso dataframe semelhante à tabela que utilizamos em nosso exemplo:

```
import pandas as pd
nome =
["Paula","Ana","Denise","Arthur","Maria","Thiago","Alan","Catarina","Marco
s","José"]
turno =
["Primeiro","Segundo","Primeiro","Segundo","Primeiro","Segundo","Primeiro"
,"Primeiro","Segundo","Primeiro"]
dataframe = pd.DataFrame({"Nome":nome,"Turno":turno})
```

Dessa forma, podemos criar a codificação *One-Hot* em nossa tabela, ficando com uma tabela similar à do nosso exemplo. Mas para que essa codificação serve? O One-Hot nos ajuda a transformar as variáveis que são categóricas (nesse caso, `Primeiro` e `Segundo`) em variáveis numéricas que são consumidas por modelos de *machine learning*.

Essa codificação que utilizamos é algo tão costumeiro de ser feito que a própria biblioteca *pandas* possui um método para fazer essa codificação, o `pd.get_dummies` :

```
turno = pd.get_dummies(dataframe[["Turno"]])
turno.head()
```

Observe que nesse método passamos como parâmetros o dataframe que queremos manipular e a coluna que queremos codificar. Veja o resultado dessa codificação:

turno_Primeiro	turno_Segundo
1	0
0	1
1	0
0	1
1	0
0	1
1	0
1	0
0	1
1	0

Ao utilizarmos o `get_dummies` da biblioteca *pandas*, podemos perceber que o nosso dataframe ganhou duas colunas, de nomes `turno_Primeiro` e `turno_Segundo` . Via de regra, a criação dos nomes é feita da seguinte forma: `nome da coluna + _ + categoria` – como você pode observar na tabela anterior.

O método que acabamos de utilizar parece bom, porém, ele tem uma falha, que diz respeito ao número de categorias que uma variável possui. Para que você consiga entender melhor esse problema da codificação *One-Hot*, vamos supor que estamos trabalhando com um dataframe de uma escola com cursos EAD no qual, além do turno em que os alunos acessam a plataforma, temos também as cidades onde esses alunos moram, conforme mostra a tabela a seguir:

Nome	Turno	Cidade
Paula	Primeiro	Curitiba
Ana	Segundo	Curitiba
Denise	Primeiro	Cascavel
Arthur	Segundo	Maringá
Maria	Primeiro	Indaiatuba
Thiago	Segundo	Cascavel
Alan	Primeiro	Cascavel
Catarina	Primeiro	Curitiba
Marcos	Segundo	Ponta Grossa
José	Primeiro	Curitiba

Caso utilizássemos a codificação *One-Hot* na coluna `Cidade` nesse exemplo específico, teríamos o total de 5 colunas a mais. Parece pouco, mas imagine que temos um total de 10 mil alunos onde temos ao menos um aluno morando em cada município do Brasil. Neste caso, teríamos 5.570 colunas a mais em nosso dataframe! Esse excesso de colunas pode não ser viável, piorando os resultados do nosso modelo de *machine learning* ou até mesmo deixando-o mais lento.

A próxima codificação que veremos consegue contornar esse problema de excesso de categorias.

7.2 Codificação de variáveis categóricas por contagem ou frequência de observação

Como dito no final da codificação *One-Hot*, vimos que, com um número elevado de categorias, podemos ter problemas de desempenho em algoritmos de *machine learning*. Então, para contornarmos esse problema, vamos utilizar as codificações por contagem e, depois, por frequência de observação.

Antes de começarmos com a nossa codificação, vamos utilizar a nossa tabela hipotética de alunos incrementada com os dados das cidades onde os alunos e alunas estão assistindo às aulas:

Nome	Turno	Cidade
Paula	Primeiro	Curitiba
Ana	Segundo	Curitiba
Denise	Primeiro	Cascavel
Arthur	Segundo	Maringá
Maria	Primeiro	Indaiatuba
Thiago	Segundo	Cascavel
Alan	Primeiro	Cascavel
Catarina	Primeiro	Curitiba
Marcos	Segundo	Ponta Grossa
José	Primeiro	Curitiba

Para iniciar a codificação por contagem, devemos contar quantas vezes cada cidade aparece em nossa tabela:

- Curitiba: 4
- Cascavel: 3
- Maringá: 1
- Indaiatuba: 1
- Ponta Grossa: 1

Depois de feita essa contagem, basta substituir o nome da cidade pelo número da contagem, conforme mostrado na seguinte lista: `[4,4,3,1,1,3,3,4,1,4]`

Dessa forma, conseguimos transformar as variáveis categóricas dispostas na coluna `Cidade` em variáveis numéricas. Porém, olhando somente para a coluna que acabamos de decodificar você consegue dizer qual dos valores número `1` representa Indaiatuba, Maringá ou Ponta Grossa? Provavelmente, não. Além de perdermos esse referencial de cidades, não conseguimos dizer ao certo se esses

valores `1` representam uma única categoria ou mais. Então, para contornarmos esse problema de identificação das categorias, podemos utilizar a codificação por frequência.

A codificação por frequência leva em consideração a proporção de cada variável categórica em nosso dataframe e, para isso, precisamos executar os seguintes passos:

1. Contar quantas vezes cada variável categórica aparece em nosso dataframe, de forma similar à codificação por contagem;
2. Dividir a contagem pelo total de dados que temos em nossa coluna.

Vamos a um exemplo utilizando o nosso dataframe com os alunos. O primeiro passo é contabilizar o número de vezes que cada uma das cidades aparece em nosso dataframe. Após essa contagem, obtemos o seguinte dataframe, similar ao resultado da codificação por contagem:

Nome	Turno	Cidade
Paula	Primeiro	4
Ana	Segundo	4
Denise	Primeiro	3
Arthur	Segundo	1
Maria	Primeiro	1
Thiago	Segundo	3
Alan	Primeiro	3
Catarina	Primeiro	4
Marcos	Segundo	1
José	Primeiro	4

O segundo passo para fazermos a codificação por frequência é dividir os valores da coluna `Cidade` pelo total de dados da coluna `Cidade`; para nós, a divisão será pelo número total de linhas, ou seja, 10. Após a divisão de cada valor da coluna `Cidade` por 10, teremos o seguinte dataframe codificado:

Nome	Turno	Cidade
Paula	Primeiro	0.4
Ana	Segundo	0.4
Denise	Primeiro	0.3
Arthur	Segundo	0.1
Maria	Primeiro	0.1
Thiago	Segundo	0.3
Alan	Primeiro	0.3
Catarina	Primeiro	0.4
Marcos	Segundo	0.1
José	Primeiro	0.4

Em um primeiro momento você pode pensar que não há diferença nenhuma entre as duas codificações que acabamos de ver, porém não é bem assim. A codificação por frequência substitui a variável categórica pela sua devida proporção; sendo assim, quando somarmos todas as proporções, teremos a totalidade do nosso dataframe representado pelo valor `1`. Veja:

```
Curitiba: 0.4
Cascavel: 0.3
Maringá: 0.1
Indaiatuba: 0.1
Ponta Grossa: 0.1


0.4+0.3+0.1+0.1+0.1 = 1
```

Veja que cada um desses valores representa uma variável categórica única, indicando que cada valor `0.1` representa uma categoria distinta. Dessa forma, podemos perceber que cada um dos valores

`0.1` representa uma categoria distinta, e não a mesma, caso contrário, teríamos um valor superior ao número `1` no somatório.

Agora que vimos como essa codificação funciona, vamos prosseguir para a codificação em Python.

```
import pandas as pd

nome =
["Paula","Ana","Denise","Arthur","Maria","Thiago","Alan","Catarina","Marco
s","José"]
turno =
["Primeiro","Segundo","Primeiro","Segundo","Primeiro","Segundo","Primeiro"
,"Primeiro","Segundo","Primeiro"]
cidade =
["Curitiba","Curitiba","Cascavel","Cascavel","Maringá","Indaiatuba","Casca
vel","Cascavel","Curitiba","Ponta Grossa"]

dataframe= pd.DataFrame({"Nome":nome,"Turno":turno,"Cidade":cidade})
```

Criado o nosso dataframe, vamos primeiro fazer a codificação por contagem:

```
contagem_cidade = dataframe["Cidade"].value_counts().to_dict()
```

Quando utilizamos o comando `dataframe["cidade"].value_counts()`, estamos fazendo a contagem de cada uma das categorias, como pode ser visto a seguir:

```
'Cascavel': 4,
 'Curitiba': 3,
 'Maringá': 1,
 'Indaiatuba': 1,
 'Ponta Grossa': 1,
```

Olhando para esse objeto `contagem_cidade` que acabamos de construir, podemos observar que ele é um objeto do tipo `dict`. Para confirmar isso, podemos utilizar o código `type(dataframe["cidade"].value_counts())`.

Então, o que precisamos fazer para prosseguir com a nossa codificação é converter esse objeto do tipo `Series` para um objeto do tipo dicionário, e para isso utilizaremos o seguinte código:

```
contagem_cidade = dict(dataframe["cidade"].value_counts())

print(contagem_cidade)
```

O resultado do código anterior pode ser visto a seguir:

```
{'Cascavel': 4, 'Curitiba': 3, 'Maringá': 1, 'Indaiatuba': 1, 'Ponta Grossa': 1}
```

Como resultado, temos um dicionário com a chave igual ao nome da cidade e o valor com a contagem de cada uma das cidades. Vamos fazer um mapeamento em nossa coluna para convertermos o nome da cidade pelo número da contagem. Para isso, utilizaremos o seguinte código:

```
dataframe["cidade"] = dataframe["cidade"].map(contagem_cidade)
```

O resultado do código anterior é a tabela a seguir:

Nome	Turno	Cidade
Paula	Primeiro	3
Ana	Segundo	3
Denise	Primeiro	4
Arthur	Segundo	4
Maria	Primeiro	1
Thiago	Segundo	1
Alan	Primeiro	4
Catarina	Primeiro	4
Marcos	Segundo	3
José	Primeiro	1

Agora vamos criar uma função para fazermos a codificação, como mostra o código a seguir:

```
def codificacao_contagem(dataframe,coluna,dicionario):
     dataframe[coluna] = dataframe[coluna].map(dicionario)
     return dataframe
```

A função que criamos `codificacao_contagem` recebe três parâmetros: o primeiro é o dataframe que desejamos codificar, o segundo é a coluna que desejamos codificar e o terceiro são os valores que passaremos como parâmetro para o método `map` para gerar a codificação.

O QUE É O MAP?

O método `map` é uma forma concisa e eficiente de aplicar uma mesma operação a todos os elementos de uma sequência, sem a necessidade de escrevermos um loop explícito. Ele permite a transformação dos elementos de uma sequência de acordo com uma função definida pelo usuário.

Dessa forma concluímos a codificação por contagem. Agora, criaremos uma função para fazermos a codificação por frequência, como no código a seguir:

```
def codificacao_frequencia(dataframe,coluna):
    aux =
(dataframe[coluna].value_counts()/len(dataframe[coluna])).to_dict()
    coluna_aux = round(dataframe[coluna].map(aux),2)

    return coluna_aux
```

E desta forma conseguiremos fazer a codificação por frequência usando a função `codificacao_frequencia` . Após executar a função `codificacao_frequencia` para a coluna cidade:

```
dataframe["Cidade"] = codificacao_frequencia(dataframe,"Cidade")
```

Teremos como resultado o seguinte dataframe:

Nome	Turno	Cidade
Paula	Primeiro	0.27
Ana	Segundo	0.27
Denise	Primeiro	0.36
Arthur	Segundo	0.36
Maria	Primeiro	0.09
Thiago	Segundo	0.09
Alan	Primeiro	0.36
Catarina	Primeiro	0.36
Marcos	Segundo	0.27
José	Primeiro	0.09

A vantagem desse tipo de codificação é que o nosso referencial não é perdido por completo, pois ao somarmos todas as frequências devemos ter o valor final igual a `1`; sendo assim, se na coluna `cidade` não tivéssemos as cidades `Indaiatuba` e `Ponta Grossa`, a cidade `Maringá` teria o valor de `0.3`, para somar `1` no total.

7.3 Codificação com números inteiros de maneira ordenada

A utilização dos algoritmos de codificação em modelos não lineares de *machine learning* geralmente resulta em um bom desempenho, mas se esses algoritmos não seguirem uma abordagem linear, a eficiência deles pode ser comprometida. Isso ocorre, pois modelos não lineares permitem uma grande flexibilidade em seus ajustes, enquanto os modelos lineares precisam seguir uma abordagem

linear. Nestes casos, devemos fazer a codificação das variáveis de modo a preservar a linearidade delas, como veremos nesta seção.

Para lidarmos com essa situação de linearidade, podemos utilizar um outro método de codificação de variável categórica. Neste método que vamos ver, atribuímos valores inteiros às categorias, porém essa distribuição não é feita de forma aleatória e, sim, seguindo alguns passos:

1. Calculamos o valor médio da meta ou variável-alvo por categoria;
2. Ordenamos de forma crescente as categorias do valor médio do alvo;
3. Atribuímos valores inteiros às categorias ordenadas, começando em `0` e indo até `k-1`, onde *k* é o número de categorias distintas.

Esses passos visam criar uma relação monotônica entre a variável categórica, que é a que desejamos codificar, e uma outra variável, que é a variável-alvo e que será utilizada como referência para distribuição dos valores inteiros.

Para esse exemplo, vamos dispensar os cálculos manuais e vamos partir para o código em Python. Vamos utilizar a tabela que usamos até agora, mas vamos adicionar uma coluna, que será a `idade`, conforme mostra a tabela a seguir:

nome	turno	cidade	idade
Paula	Primeiro	Curitiba	18
Ana	Segundo	Curitiba	18
Denise	Primeiro	Cascavel	17
Arthur	Segundo	Maringá	19
Maria	Primeiro	Indaiatuba	21
Thiago	Segundo	Cascavel	22
Alan	Primeiro	Cascavel	23
Catarina	Primeiro	Curitiba	25
Marcos	Segundo	Ponta Grossa	18
José	Primeiro	Curitiba	17

Com essa tabela, vamos criar um dataframe, com o seguinte código:

```
nome =
["Paula","Ana","Denise","Arthur","Maria","Thiago","Alan","Catarina","Marcos","José"]
turno =
["Primeiro","Segundo","Primeiro","Segundo","Primeiro","Segundo","Primeiro","Primeiro","Segundo","Primeiro"]
cidade =
["Curitiba","Curitiba","Cascavel","Maringá","Indaiatuba","Cascavel","Cascavel","Curitiba","Ponta Grossa","Curitiba"]
idade = [18,18,17,19,21,22,23,25,18,17]

dataframe =
pd.DataFrame({"Nome":nome,"Turno":turno,"Cidade":cidade,"Idade":idade})
```

Vamos construir um gráfico que exibe a relação entre a coluna `cidade` e a coluna `idade` :

```
def grafico(dataframe,coluna1,coluna2):
  plt.figure(figsize=(12,7))
  dataframe.groupby(coluna1)[coluna2].mean().plot()
  texto = f"Relação entre a {coluna1} e a média da {coluna2}"
  plt.title(texto)
  plt.ylabel(coluna2)
```

Para utilizarmos essa função, basta usarmos o comando `grafico(dataframe,"cidade","idade")` e o resultado será similar ao da figura a seguir:

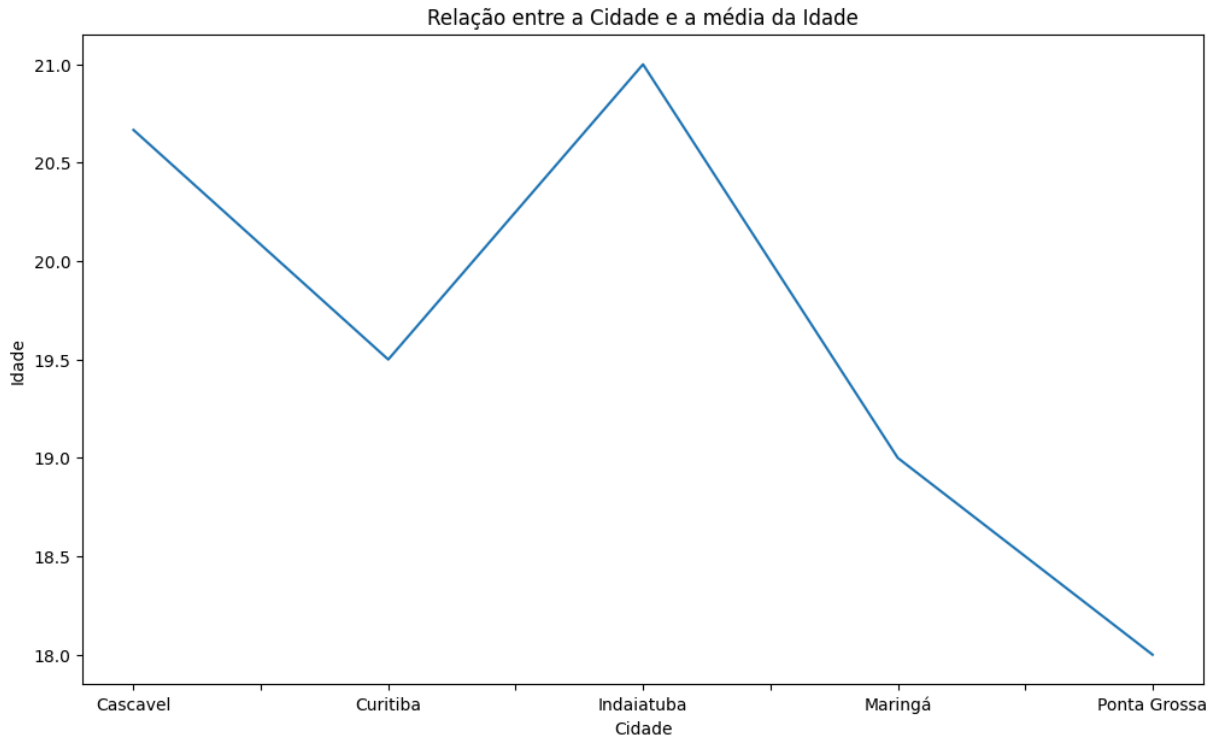

Figura 7.1: Gráfico exibindo o resultado das médias das idades por cidades

Perceba que o gráfico tem inclinações abruptas: isso é o que chamamos de não monotonicidade. Dizemos que uma função é monotônica quando ela preserva ou inverte a relação de ordem. Quando ela preserva a relação de ordem, dizemos que é monotônica crescente; quando inverte a relação de ordem, dizemos que é monotônica decrescente.

Podemos perceber que o gráfico não preserva essas relações, pois ora cresce, ora decresce. Então, o que faremos é codificar os valores da coluna `cidade` para transformarmos essa relação em uma relação monotônica.

Para construirmos essa relação, vamos começar criando um agrupamento das nossas cidades utilizando a média das idades em ordem crescente com o código a seguir:

```
rotulos_ordenados = dataframe.groupby(["Cidade"])
["Idade"].mean().sort_values().index
```

O resultado pode ser visto a seguir:

```
Index(['Ponta Grossa', 'Curitiba', 'Cascavel', 'Maringá', 'Indaiatuba'],
dtype='object', name='cidade')
```

Você pode se perguntar: por que a lista que criamos possui esta ordem? Se olharmos a média das idades, temos algo semelhante a:

```
Ponta Grossa    17.0
Curitiba        18.0
Cascavel        21.0
Maringá         21.0
Indaiatuba      22.0
```

Ou seja, `Ponta Grossa` possui a menor média de idade, que é de 17 anos; em seguida, temos `Curitiba` , com 18 anos, e assim por diante, até Indaiatuba, com 22 anos.

Vamos transformar a lista que criamos em um dicionário Python onde a chave é o nome da cidade e o valor é o índice da nossa lista anterior:

```
ordinal = {k:i for i, k in enumerate(rotulos_ordenados,0)}
```

O dicionário que criamos com o código anterior pode ser visto a seguir:

```
{'Ponta Grossa': 0,
 'Curitiba': 1,
 'Cascavel': 2,
 'Maringá': 3,
 'Indaiatuba': 4}
```

Com o dicionário criado, podemos fazer um mapeamento em nosso dataframe para substituirmos o nome das cidades pelos valores do nosso dicionário, como mostra o código a seguir:

```
dataframe["cidade"] = dataframe["cidade"].map(ordinal)
```

Dessa forma, criamos a nossa codificação com números inteiros de forma ordenada. Por fim, podemos criar o gráfico para observarmos a relação da cidade com a idade; para isso, basta chamarmos a nossa função `grafico`:

```
grafico(dataframe,"cidade","idade")
```

E o resultado desse código pode ser visto na figura a seguir:

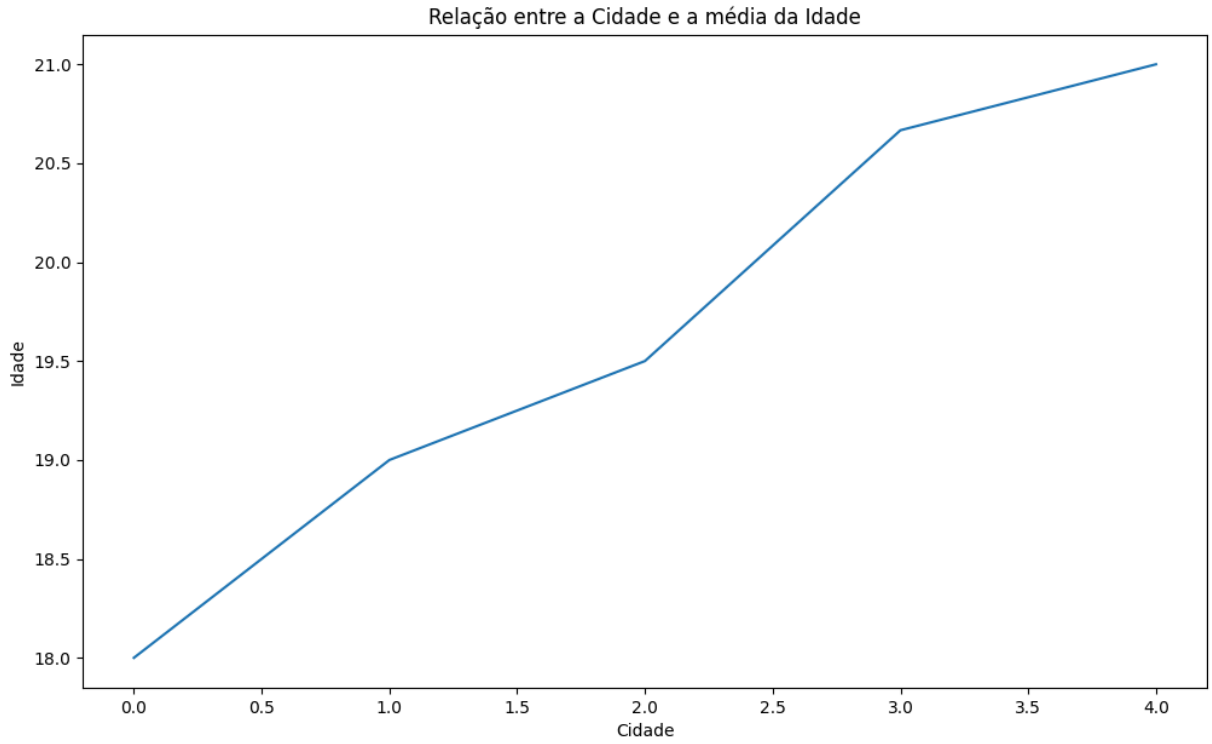

Figura 7.2: Gráfico exibindo o resultado das cidades com os rótulos ordenados e a média das idades

Atente-se ao fato de que os números mostrados no dicionário não se referem à incidência das cidades em nosso dataframe, mas sim à sua posição com base na média das idades. Por exemplo, a cidade que está posicionada como a de menor média das idades é Ponta Grossa, enquanto a cidade que está posicionada como a de maior média das idades é Indaiatuba.

Com isso concluímos mais uma parte e agora conseguiremos trabalhar com valores que não são numéricos com a codificação. No próximo capítulo, veremos um projeto prático para fixarmos melhor esse conteúdo.

Capítulo 8
Projeto prático — Codificação de variáveis categóricas

Neste projeto prático, voltaremos a operar com o nosso conjunto de dados do desastre do Titanic para fazermos a codificação de variáveis categóricas das colunas `Sex` e `Embarked`. Vamos começar importando a biblioteca *pandas* e carregando o `projeto2.csv`, o mesmo arquivo `csv` que utilizamos no projeto prático do segundo capítulo. Vamos utilizá-lo porque ele contém um dataframe sem valores ausentes.

Vamos carregar os nossos dados utilizando o código a seguir:

```
import pandas as pd

dataframe = pd.read_csv("projeto2.csv")
dataframe.drop(columns = ["Unnamed: 0"],inplace=True)
```

Diferente do último projeto prático, em que utilizamos apenas a biblioteca *scikit-learn*, neste vamos utilizar a também a *feature-engine* para fazermos as codificações. Antes de começarmos a discutir os processos de codificação de valores categóricos, vamos dividir o nosso conjunto em dados de treino e dados de teste com o código a seguir:

```
from sklearn.model_selection import train_test_split

X_treino, X_teste, y_treino, y_teste =
train_test_split(dataframe.drop("Survived",axis =
1),dataframe["Survived"],test_size = 0.3,random_state = 0)
```

O resultado dessa codificação é um dataframe similar à tabela a seguir, dividida em duas para melhor visualização, para os dados do dataframe `X_treino`:

PassagerId	Pclass	Name	Sex	Age
988	1	Cavendish, Mrs.	female	76.0
1273	3	Foley, Mr. Joseph	male	26.0
981	2	Wells Master	male	2.0

PassagerId	SibSp	Parch	Ticket	Fare	Embarked
988	1	0.0	19877	78.8500	S
1273	0	0.0	330910	7.6792	Q
981	1	1.0	29103	23.0000	S

E para os valores de teste:

PassengerId	Pclass	Name	Sex	Age
1252	3	Master William Henry	male	14.5
1062	3	Mr. Simon	male	27.0
1116	1	Mrs. Edward	female	53.0
1250	3	Mr. Patrick	male	27.0
1201	3	Mrs. Claus Peter	female	45.0

PassengerId	SibSp	Parch	Ticket	Fare	Embark
1252	8	2.0	CA. 2343	69.55	S
1062	0	0.0	S.O./P.P. 251	7.55	S
1116	0	0.0	PC 17606	27.4458	C
1250	0	0.0	368402	7.75	Q
1201	1	0.0	350026	14.1083	S

Os valores que vamos manipular neste projeto prático são os referentes ao gênero na coluna `Sex` e os referentes ao local de embarque na coluna `Embarked`.

8.1 Codificação One-Hot utilizando a biblioteca scikit-learn

Veremos como fazer a codificação *One-Hot* utilizando a biblioteca *scikit-learn*. Para isso, vamos importar a classe `OneHotEncoder` da biblioteca *scikit-learn*:

```
from sklearn.preprocessing import OneHotEncoder
```

> **One-Hot e OneHotEncoder**
>
> *One-Hot*: nome dado ao método de codificação. O `OneHotEncoder` é o método utilizado pela biblioteca `sklearn` para fazer a codificação do tipo *One-Hot*.

Vamos instanciar o nosso `OneHotEncoder`:

```
ohe = OneHotEncoder(categories="auto",drop="first",sparse=False)
```

O método `OneHotEncoder` foi instanciado com os parâmetros `categories` como `auto` e `drop` como `first`. O primeiro faz com que o codificador aprenda quais as categorias a serem codificadas, já o segundo é mostrado na seguinte lista: `[male, female, male, male]`.

O resultado da codificação *One-Hot* pode ser visto na tabela a seguir:

Sex	male	female
male	1	0
female	0	1
male	1	0
male	1	0

Vamos desconsiderar a primeira coluna `male` do *One-Hot*, ficando com a seguinte tabela:

Sex	female
male	0
female	1
male	0
male	0

Observe que ainda somos capazes de identificar o gênero das pessoas em nossa tabela. Isso ocorre porque, se o valor da coluna `female` for igual a `0`, então com certeza se trata do gênero `male` (e vice-versa), já que nesse exemplo estamos trabalhando com apenas dois gêneros.

O método `drop = "first"` remove a primeira coluna para reduzir o número de colunas retornadas pelo método `OneHotEncoder`, enquanto o método `sparse = False` faz com que seja retornada apenas a matriz de valores codificados.

Vamos ajustar os dados com o método `.fit` passando os dados que queremos codificar com o código a seguir:

```
ohe.fit(X_treino[["Sex","Embarked"]])
```

Vamos utilizar o método `transform` para transformar os nossos dados de treino e de teste:

```
treino_onehot = ohe.transform(X_treino[["Sex","Embarked"]])
teste_onehot = ohe.transform(X_teste[["Sex","Embarked"]])
```

Vale lembrar que a transformação utilizando a biblioteca *scikit-learn* transforma o dataframe em um `numpy array`, porém aqui não converteremos para um dataframe como sempre fazemos. Para você entender por que não faremos isso, observe a tabela a seguir:

```
treino_onehot
```

```
[1.,0.,1.],
[1.,0.,1.],
[0.,0.,1.],
[0.,0.,0.],
[1.,0.,1.],
[1.,0.,1.],
```

Olhando para a tabela anterior, conseguimos perceber que a transformação *One-Hot* utilizando o `OneHotEncoder` tira de nós o referencial. Por exemplo, sabemos que a primeira coluna diz respeito ao gênero da coluna `Sex`, porém olhando somente para essa coluna não temos como saber se ela representa o gênero masculino ou feminino. Para a coluna `Embarked`, sabemos que temos duas categorias, pois a primeira coluna foi retirada devido ao parâmetro `drop = first`; da mesma forma, olhando somente para essas

colunas, não temos como saber se elas representam a categoria `Q`, `S` ou `C`.

Claro que, neste caso, poderíamos analisar o `numpy array` codificado e o dataframe para descobrir o que cada uma dessas colunas representa. Porém, imagine se, em vez de 3 colunas, tivéssemos 25? Perderíamos muito tempo com nossa análise para imputar os nomes corretos a cada coluna, então nesse caso não vale a pena fazer esse processo de conversão para um dataframe.

8.2 Fazendo a codificação One-Hot com a biblioteca feature-engine

Vimos que na codificação *One-Hot* utilizando a biblioteca *scikit-learn* perdemos o referencial do que cada coluna representa após a codificação. Porém, isso não acontece quando utilizamos a biblioteca *feature-engine*.

Vamos importar a classe `OneHotEncoder` da biblioteca *feature-engine*, lembrando que, caso esteja utilizando o *Google Colaboratory*, você terá que instalar a biblioteca *feature-engine* com o comando `!pip install feature-engine`.

```
from feature_engine.encoding import OneHotEncoder
```

Vamos instanciar a classe que acabamos de importar da seguinte forma:

```
ohe_fe = OneHotEncoder(top_categories = None,variables =
["Sex","Embarked"],drop_last=True)
```

O parâmetro `top_categories`, em que passamos `None`, indica que queremos que todas as categorias sejam codificadas, mas poderíamos passar um número inteiro também para indicarmos quantas categorias desejamos codificar. O parâmetro `variables`

indica as colunas que queremos codificar, no nosso caso são as colunas `Sex` e `Embarked`. E o `drop_last` igual a `True` faz o equivalente ao `drop_first`, porém, em vez de excluir a primeira coluna, exclui a última.

Vamos ajustar o nosso modelo passando os dados de treino da seguinte forma:

```
ohe_fe.fit(X_treino)
```

Não precisamos passar as colunas que queremos codificar, pois já as passamos no parâmetro `variables`. Podemos transformar os nossos dados com o seguinte código:

```
treino_ohe_fe = ohe_fe.transform(X_treino)
teste_ohe_fe = ohe_fe.transform(X_teste)
```

O resultado dessa codificação pode ser visto nas tabelas a seguir. Para os dados de treino `treino_ohe_fe`, temos:

PassengerId	Pclass	Name	Age	SibSp	Parch
988	1	Cavendish, Mrs. Tyrell William (Julia Florence Siegel)	76.0	1	0.0
1273	3	Foley, Mr. Joseph	26.0	0	0.0
981	2	Wells, Master. Ralph Lester	2.0	1	1.0
1125	3	Linehan, Mr. Michael	27.0	0	0.0
1083	1	Salomon, Mr. Abraham L.	27.0	0	0.0

PassengerId	Fare	Sex_female	Embarked_S	Emba
988	78.85	1	1	0
1273	7.8792	0	0	1
981	23.0	0	1	0
1125	7.8792	0	0	1
1083	26.0	0	1	0

E, para os dados de teste, temos:

PassengerId	Pclass	Name	Age	SibSp	Parch	Ti
1252	3	William Henry	14.5	8	2.0	CA. 234
1062	3	Simon	27.0	0	0.0	S.O 251
1116	1	Edward	53.0	0	0.0	PC 176
1250	3	Patrick	27.0	0	0.0	368
1201	3	Claus Peter	45.0	1	0.0	350

PassengerId	Fare	Sex_female	Embarked_S	Emba
1252	69.55	0	1	0
1062	7.55	0	1	0
1116	27.4458	1	0	0
1250	7.75	0	0	1
1201	14.1083	1	1	0

Observe que, nessa transformação, não houve conversão do dataframe para um `numpy array`, ele continua sendo um `pandas dataframe`. Mas as colunas `Sex` e `Embarked` foram substituídas pelas colunas `Sex_female`, `Embarked_S` e `Embarked_Q`. Desta forma, conseguimos manter o contexto das colunas, o que pode ser útil em algumas análises ou até mesmo na compreensão dos dados que estão entrando na *pipeline* dos algoritmos de *machine learning*.

8.3 Substituindo categorias por contagem ou frequência de observações

A biblioteca *scikit-learn* não possui uma classe para fazer a codificação de categorias por contagem ou frequência de observações; sendo assim, utilizaremos somente a biblioteca *feature_engine* nesta seção. Vamos começar importando a classe `CountFrequencyEncoder` da biblioteca *feature-engine*:

```
from feature_engine.encoding import CountFrequencyEncoder
```

Vamos instanciar nossa classe. O primeiro método que veremos é o da codificação por contagem de categoria nas colunas `Sex` e `Embarked`:

```
contagem = CountFrequencyEncoder(encoding_method="count",variables=
["Sex","Embarked"])
```

Agora, vamos ajustar os dados com nosso conjunto de treino, da seguinte forma:

```
contagem.fit(X_treino)
```

Podemos inspecionar os valores aprendidos pela nossa classe `CountFrequencyEncoder` utilizando o comando `contagem.encoder_dict_`, e o resultado pode ser visto a seguir:

```
contagem.encoder_dict_
{'Sex':{'male':197,'female':95},'Embarked':{'S':194,'C':66,'Q':32}}
```

Por fim, vamos fazer a codificação por contagem utilizando o método `transform`:

```
treino_contagem = contagem.transform(X_treino)
teste_contagem = contagem.transform(X_teste)
```

O resultado dessa codificação pode ser observado nas tabelas a seguir. Para os dados de treino:

```
treino_contagem.head(3)
```

PassengerId	Pclass	Name	Sex	Age
988	1	Cavendish, Mrs. Tyrell William (Julia Florence Siegel)	95	76.0
1273	3	Foley, Mr. Joseph	197	26.0
981	2	Wells, Master. Ralph Lester	197	2.0

PassengerId	SibSp	Parch	Ticket	Fare	Embarked
988	1	0.0	19877	78.85	194
1273	0	0.0	330910	7.8792	32
981	1	1.0	29103	23.0	194

E para os dados de teste:

```
teste_contagem.head(3)
```

PassengerId	Pclass	Name	Sex	Age
1252	3	Sage, Master. William Henry	197	14.5
1062	3	Lithman, Mr. Simon	197	27.0
1116	1	Candee, Mrs. Edward (Helen Churchill Hungerford)	95	53.0

PassengerId	SibSp	Parch	Ticket	Fare	Embark
1252	8	2.0	CA. 2343	69.55	194

PassengerId	SibSp	Parch	Ticket	Fare	Embark
1062	0	0.0	S.O./P.P. 251	7.55	194
1116	0	0.0	PC 17606	27.4458	66

Para fazermos a codificação pela frequência, basta substituirmos o parâmetro `encoding_method` , que está como `count` , por `frequency` , da seguinte forma:

```
frequencia = CountFrequencyEncoder(encoding_method = "frequency",variables = ["Sex","Embarked"])
```

Basta ajustarmos com os dados de treino:

```
frequencia.fit(X_treino)
```

Podemos inspecionar os valores aprendidos na etapa de ajuste com o comando `frequencia.encoder_dict_` , e o resultado pode ser visto no código a seguir:

```
{'Sex': {'male': 0.6746575342465754, 'female': 0.3253424657534247},
 'Embarked': {'S': 0.6643835616438356,
  'C': 0.22602739726027396,
  'Q': 0.1095890410958904}}
```

Por fim, vamos transformar os nossos dados de treino e de teste com o seguinte código:

```
treino_frequencia = frequencia.transform(X_treino)
teste_frequencia = frequencia.transform(X_teste)
```

E o resultado pode ser visto nas tabelas a seguir. Para os dados de treino:

```
treino_contagem.head(3)
```

PassengerId	Pclass	Name	Sex	Age
988	1	Tyrell William	0.32	76.0
1273	3	Joseph	0.67	26.0
981	2	Master. Ralph Lester	0.67	2.0

PassengerId	SibSp	Parch	Ticket	Fare	Embarked
988	1	0.0	19877	78.85	0.66
1273	0	0.0	330910	7.8792	0.10
981	1	1.0	29103	23.0	0.66

Para os dados de teste:

```
teste_contagem.head(3)
```

PassengerId	Pclass	Name	Sex	Age
1252	3	William Henry	0.67	14.5
1062	3	Simon	0.67	27.0
1116	1	Edward	0.32	53.0

PassengerId	SibSp	Parch	Ticket	Fare	Embark
1252	8	2.0	CA. 2343	69.55	0.66
1062	0	0.0	S.O./P.P. 251	7.55	0.66
1116	0	0.0	PC 17606	27.4458	0.22

8.4 Codificação com inteiros de maneira ordenada utilizando a biblioteca feature-engine

Para fazermos esse tipo de codificação, utilizaremos a classe `OrdinalEncoder` da biblioteca *feature-engine*. Vamos fazer a seguinte importação:

```
from feature_engine.encoding import OrdinalEncoder
```

Vamos instanciar nossa classe da seguinte forma:

```
ordinal = OrdinalEncoder(encoding_method="ordered", variables=
["Sex","Embarked"])
```

O parâmetro `encoding_method` indica à nossa classe qual o tipo de codificação que desejamos fazer. Neste caso, queremos fazer a codificação de forma ordenada (`"ordered"`), porém poderíamos passar o valor `"arbitrary"` para fazermos a codificação de números inteiros de forma arbitrária. O segundo parâmetro, `variables` , indica à nossa classe quais são as colunas em que desejamos fazer a codificação.

Podemos ajustar o modelo passando os dados de treino no parâmetro `fit` . Entretanto, é importante destacar que a codificação ordinal é usada para criar uma relação monotônica entre a nossa variável-alvo e a variável que desejamos codificar. Em nosso caso, a variável-alvo é a `Survived` , que está armazenada nas colunas `y_treino` e `y_teste` . Portanto, para ajustarmos o modelo, devemos passar esses dois parâmetros da seguinte forma:

```
ordinal.fit(X_treino, y_treino)
```

Esse código ajusta os dados da variável `y_treino` com base nos dados da variável `X_treino` . Dessa forma, estabelecemos a relação entre as variáveis que vai permitir que o modelo aprenda a fazer as previsões com base nos dados usados para o treino.

Podemos inspecionar os dados aprendidos na etapa de ajuste com o comando `ordinal.encoder_dict_` e o resultado pode ser visto a seguir:

```
ordinal.encoder_dict_
```

```
{'Embarked':{'C':0,'Q':2,'S':1},'Sex':{'female':1,'male':0}}
```

Vale lembrar que os valores apresentados no dicionário anterior não estão relacionados ao número de ocorrências no dataframe, mas sim à ordem em que eles aparecem. Por exemplo, no caso de `Embarked`, a categoria `C` é representada como `0`, indicando que é a que aparece em primeiro lugar na ordem, enquanto a categoria `S` é representada pelo valor `2`, indicando que é a que aparece em terceiro lugar na ordem.

Por fim, podemos transformar nossos dados com o comando `transform`, como no código a seguir:

```
treino_ordinal = ordinal.transform(X_treino)
teste_ordinal = ordinal.transform(X_teste)
```

O resultado dessa transformação pode ser visto a seguir. Para os dados de treino:

```
treino_ordinal.head(3)
```

PassengerId	Pclass	Name	Sex	Age
988	1	Tyrell William	1	76.0
1273	3	Mr. Joseph	0	26.0
981	2	Master. Ralph Lester	0	2.0

PassengerId	SibSp	Parch	Ticket	Fare	Embarked
988	1	0.0	19877	78.85	1
1273	0	0.0	330910	7.8792	2
981	1	1.0	29103	23.0	1

E para os dados de teste:

```
teste_contagem.head(3)
```

PassengerId	Pclass	Name	Sex	Age
1252	3	William Henry	0	14.5
1062	3	Simon	0	27.0
1116	1	Edward	1	53.0

PassengerId	SibSp	Parch	Ticket	Fare	Embark
1252	8	2.0	CA. 2343	69.55	1
1062	0	0.0	S.O./P.P. 251	7.55	1
1116	0	0.0	PC 17606	27.4458	0

Vamos utilizar a função `grafico` que criamos no capítulo anterior para criar nosso gráfico, dessa forma poderemos observar a relação entre a variável-alvo, ou seja, a variável `Survived`, e as variáveis antes e depois da codificação. Isso nos permitirá analisar a relação entre a variável-alvo e as colunas usadas no treinamento. As duas colunas em questão são `Sex` e `Age`. Vamos relembrar o código que construímos para criar o gráfico:

```
def grafico(dataframe,coluna1,coluna2):
 plt.figure(figsize=(12,7))
 dataframe.groupby(coluna1)[coluna2].mean().plot()
 texto = f"Relação entre a {coluna1} e a média do {coluna2}"
 plt.title(texto)
 plt.ylabel(coluna2)
```

Para criar este gráfico, primeiramente concatenaremos as colunas selecionadas com a variável `Survived`. Este passo é importante para visualizarmos o impacto da substituição de variáveis categóricas por suas frequências. Além disso, ao concatenar os dados de treino e teste, poderemos analisar a proporção de cada conjunto nos dados de treino e teste. A concatenação será realizada da seguinte maneira.

```
sex_treino = X_treino["Sex"]
embarked_treino = X_treino["Embarked"]

temp = pd.concat([sex_treino,embarked_treino, y_treino], axis=1)
```

O resultado dessa concatenação pode ser visto a seguir:

```
temp.head()
```

Sex	Embarked	Survived
female	S	1
male	Q	0
male	S	0
male	Q	0
male	S	0

Vamos construir o gráfico da relação entre a média da variável-alvo e a coluna `Embarked` antes da codificação. Para isso, basta executarmos a seguinte função que criamos anteriormente: `grafico(temp,"Embarked","Survived")`. O resultado pode ser visto na figura a seguir:

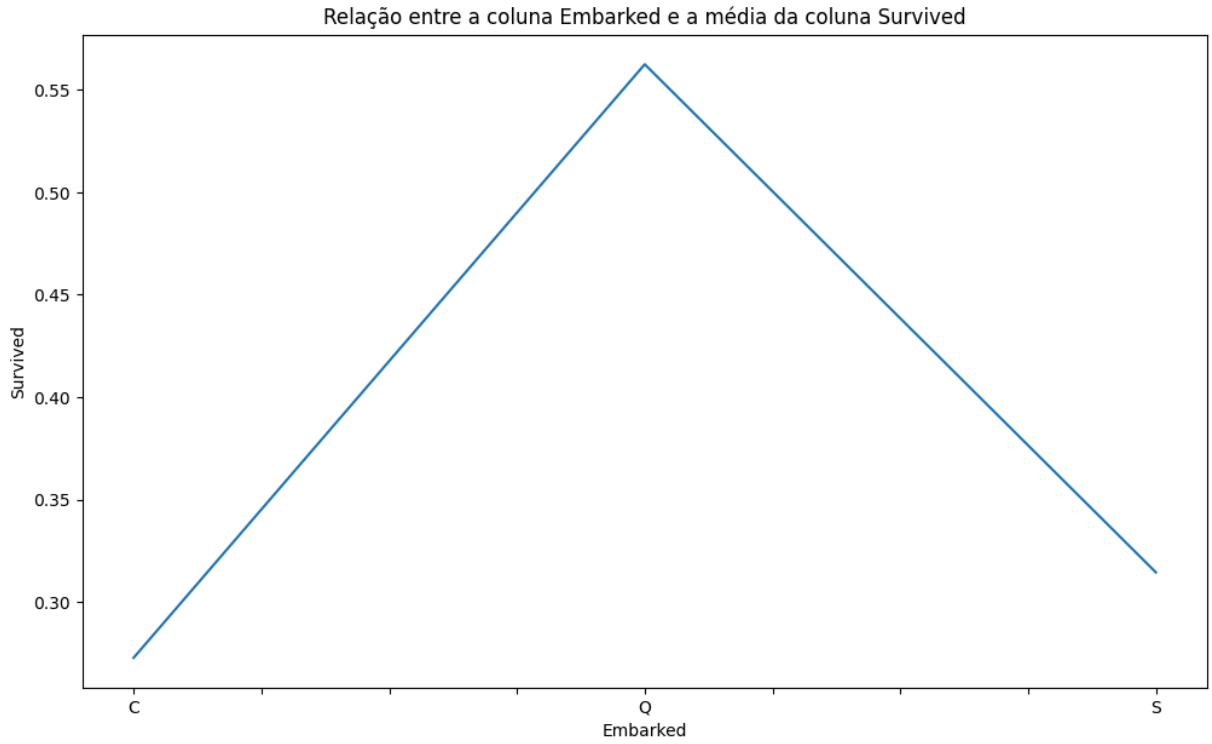

Figura 8.1: Relação entre a média da coluna "Embarked" e a coluna "Survived".

Vamos criar uma função que cria uma função para exibirmos a relação entre as variáveis `Sex` e `Survived` e `Embarked` e `Survived`:

```
def concatena_e_plota(dataframe,alvo,coluna1,coluna2):
    aux1 = dataframe[coluna1]
    aux2 = alvo

    temp = pd.concat([aux1,aux2],axis=1)
    grafico(temp,coluna1,coluna2)
```

A função que acabamos de criar faz a seguinte execução:

1. Armazena em `aux1` o conteúdo especificado pela `coluna1` do `dataframe`;
2. Armazena em `aux2` o conteúdo do `alvo`;
3. Concatena as duas series (`aux1` e `aux2`) ao longo do eixo 1 (colunas) usando a função `pd.concat`, resultando em um novo dataframe chamado de `temp`;

4. Chama a função `grafico` passando o dataframe `temp`, a `coluna1` como o eixo x e a `coluna2` como eixo y para plotar um gráfico com esses dados.

Agora, vamos utilizar essa função criada para exibir a relação entre as variáveis `Embarked` depois da codificação e a coluna `Survived`. Para isso, vamos chamar a função `concatena_e_plota` da seguinte forma:

```
concatena_e_plota(treino_ordinal,y_treino,"Embarked","Survived")
```

O resultado pode ser visto na imagem a seguir:

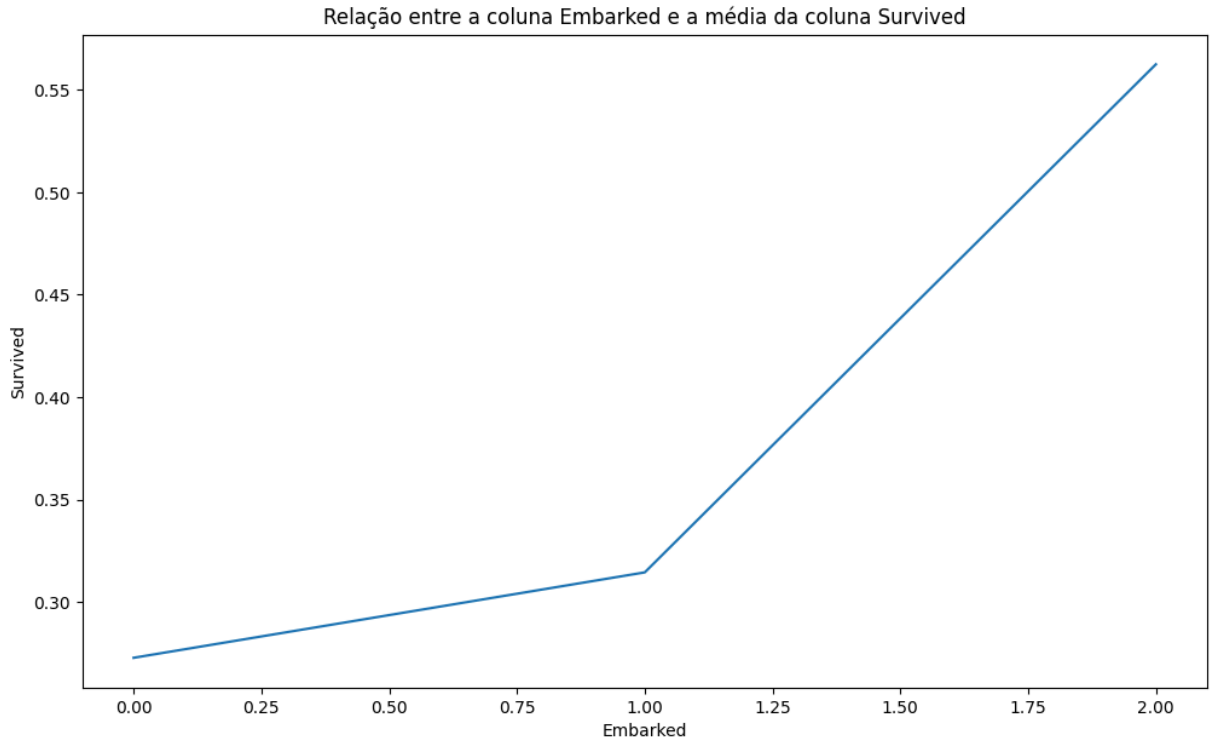

Figura 8.2: Relação entre a média da coluna "Embarked" e a coluna "Survived".

Como podemos observar o antes e depois da codificação, podemos perceber que agora temos uma relação monótona. Como já falamos no capítulo anterior, criar relações assim entre as variáveis auxilia algoritmos de *machine learning* que são linearmente dependentes. Para finalizar, vamos plotar mais um gráfico para vermos a relação entre a coluna `Sex` após a codificação e a coluna `Survived`:

```
concatena_e_plota(treino_ordinal,y_treino,"Sex","Survived")
```

O resultado pode ser visto na imagem a seguir:

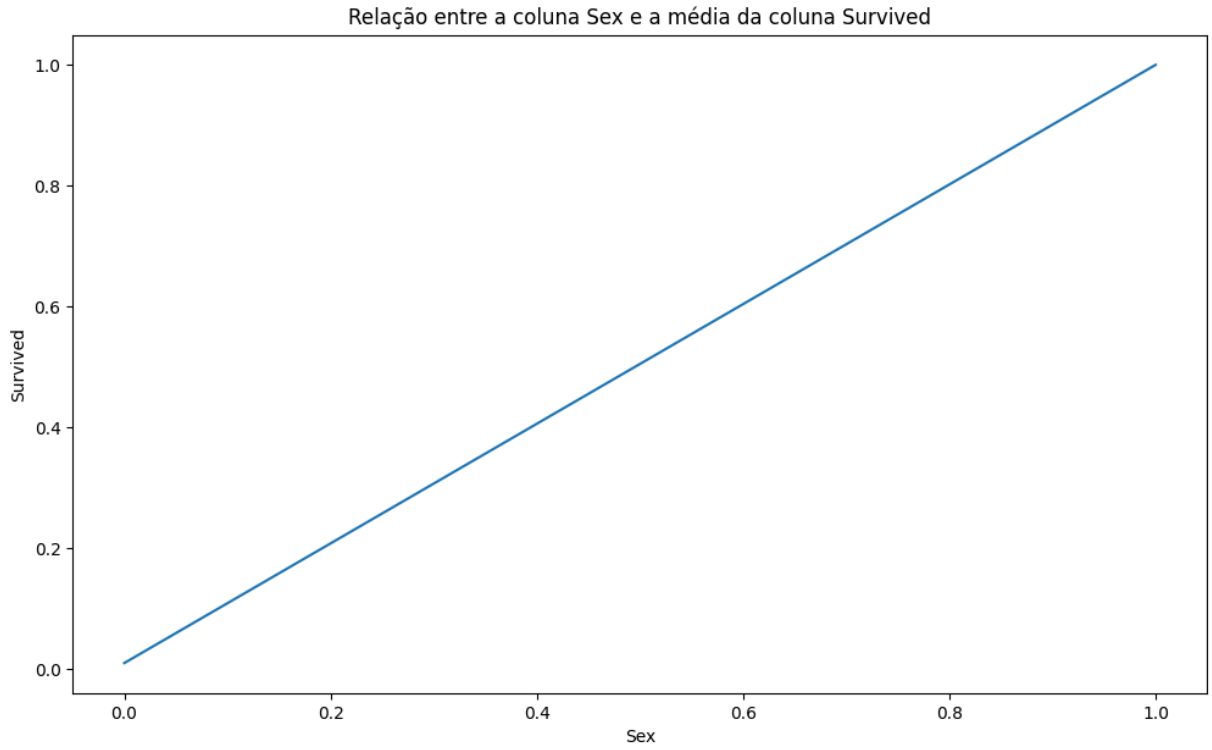

Figura 8.3: Relação entre a média da coluna "Sex" e a coluna "Survived".

Com isso, finalizamos o capítulo sobre codificações. Exploramos o uso das bibliotecas *scikit-learn* e a *feature-engine* para fazermos diversos tipos de codificações de variáveis categóricas em variáveis numéricas. Ao dominarmos essas técnicas de codificação, adquirimos habilidades essenciais para lidar com dados categóricos em problemas de aprendizado de máquina.

PARTE 5: Redução de dimensionalidade, upsampling, downsampling e métricas para avaliação

CAPÍTULO 9
Trabalhando com redução de dimensionalidade, upsampling e downsampling

> O notebook deste capítulo e do projeto prático podem ser encontrados em: materiais da parte 5 (https://github.com/ThiagoBenevides/feature-engine-transformando-dados-brutos-em-recursos/tree/main/parte_cinco)

A redução de dimensionalidade é uma técnica importante no processamento de dados, especialmente quando lidamos com grandes quantidades de dados que possuem muitas variáveis. Uma das técnicas mais populares quando falamos em redução de dimensionalidade é o PCA (*Principal Component Analysis*).

O PCA é uma técnica matemática que busca simplificar a análise de conjunto de dados complexos, reduzindo sua dimensionalidade. Para entendermos como ele funciona, é importante compreender a diferença entre colunas linearmente dependentes e independentes. Quando duas ou mais colunas de um conjunto de dados são linearmente dependentes, significa que uma pode ser expressa como combinação linear das outras. Por outro lado, as colunas são linearmente independentes quando não há relação linear entre elas.

O método PCA que veremos neste capítulo busca identificar as colunas que possuem alta correlação entre si e descarta as que possuem alta correlação com outras colunas. Isso resulta em um novo conjunto de dados com colunas linearmente independentes. Essa redução de dimensionalidade pode melhorar o desempenho de algoritmos de *machine learning* e diminuir o tempo necessário para treiná-los.

9.1 Reduzindo o tamanho do nosso conjunto com PCA

Para o nosso projeto, utilizaremos um conjunto de dados fictício sobre exame de sangue utilizado para identificar doenças. Podemos observar o nosso dataframe a seguir:

index	hemoglobina	leucocitos	plaquetas	glicose	v
0	13.74	8152.65	369081.19	147.90	3
1	19.50	7833.45	418672.55	136.78	1
2	17.31	4205.13	200187.15	107.46	4
3	15.98	6729.98	349949.63	70.84	3
4	11.56	7041.56	328698.39	134.51	1

potassio	calcio	creatinina	ureia	colesterol	diagnos
4.58	10.32	0.53	36.17	145.19	Diabetes
5.16	9.55	0.52	13.20	231.28	Hipercal
3.75	9.94	1.23	19.69	132.84	Disfunçã Renal
3.75	9.37	0.81	40.94	111.19	Insuficié Renal
4.61	9.76	1.29	31.14	124.20	Disfunçã Renal

Para este projeto, vamos reduzir o número de colunas que serão utilizadas para prever o diagnóstico. A técnica de PCA que vamos utilizar é útil em vários casos, e na área da saúde não é diferente. No caso de um exame de sangue, que normalmente possui diversas variáveis medidas, o PCA pode ser utilizado para reduzir a dimensão dos dados sem perder muita informação.

Em nosso caso, o dataframe possui 10 colunas que representam diferentes variáveis medidas em um exame de sangue e que podem estar relacionadas a diferentes doenças. Utilizando o PCA, é possível reduzir a dimensão do dataframe para um número menor de

colunas, mantendo as informações mais relevantes para identificarmos as doenças em questão.

Além disso, a redução de dimensionalidade pode ter vantagens práticas, como a redução do tempo necessário para a análise e processamento dos dados, além da possibilidade de coletar menos sangue dos pacientes para realizar o exame.

Outra vantagem da utilização do PCA é que ele pode ajudar a identificar quais variáveis são mais relevantes para a identificação de determinada doença, o que pode ajudar a focar o exame em menos variáveis, reduzindo ainda mais o tempo e custo do processo.

Entretanto, antes de reduzirmos os nossos dados, precisamos verificar se é possível fazer essa redução, ou seja, precisamos determinar se temos colunas que são linearmente dependentes de outras colunas. Para isso, podemos usar um método bem conhecido por analistas de dados popularmente chamado de **método do cotovelo**, ou, em inglês, *elbow method*. Esse método é usado para determinar o número ideal de colunas para nossos dados e, para isso, ele se baseia na ideia de que adicionar mais colunas não trará melhorias significativas no nosso algoritmo de *machine learning*.

Vamos preparar nosso conjunto de dados para utilizarmos o método do cotovelo. O que faremos é separar os dados em duas variáveis, chamadas de `X` e `y`, que armazenarão os seguintes dados:

- `X`: 'hemoglobina', 'leucocitos', 'plaquetas', 'glicose', 'vitamina_d', 'potassio', 'calcio', 'creatinina', 'ureia', 'colesterol';
- `y`: 'diagnostico'.

Para fazermos essa separação, vamos usar o seguinte código:

```
X = dados.iloc[:, :-1].values
y = dados.iloc[:, -1].values
```

Para utilizar o método do cotovelo, precisamos realizar uma análise de clusterização com diferentes números de clusters e gerar um gráfico que relacione esse número com a medida de qualidade da clusterização, geralmente representada pelo valor da soma dos quadrados das distâncias dos pontos ao centro do cluster. O gráfico deve apresentar uma curva que se assemelha a um cotovelo, e o "ponto do cotovelo" é o número ideal de clusters.

```
from sklearn.cluster import KMeans

X = dados.iloc[:, :-1].values
max_clusters = 10

sse = []

for n_clusters in range(1, max_clusters):
    kmeans = KMeans(n_clusters=n_clusters, init='k-means++',
random_state=42)
    kmeans.fit(X)
    sse.append(kmeans.inertia_)
```

Nesse método do cotovelo, utilizamos algumas palavras que podem ser novas para você, então vamos explicá-las:

- `sse` : abreviação do termo em inglês *Sum Squared Error*, ou seja, a soma do quadrado dos erros. Dentro da lista `sse` , armazenamos o erro;
- `K` : representa o número de clusters que vamos analisar;
- `kmeans.inertia_` : calcula o `sse` da clusterização realizada pelo algoritmo `k-means` .

Vamos criar o gráfico para podermos analisar nossa descoberta feita a partir do método do cotovelo:

```
plt.plot(range(1, max_clusters), sse, marker='o')
plt.title('Método do cotovelo')
plt.xlabel('Número de clusters')
```

```
plt.ylabel('Soma dos erros quadráticos (SSE)')
plt.show()
```

O resultado é o gráfico a seguir:

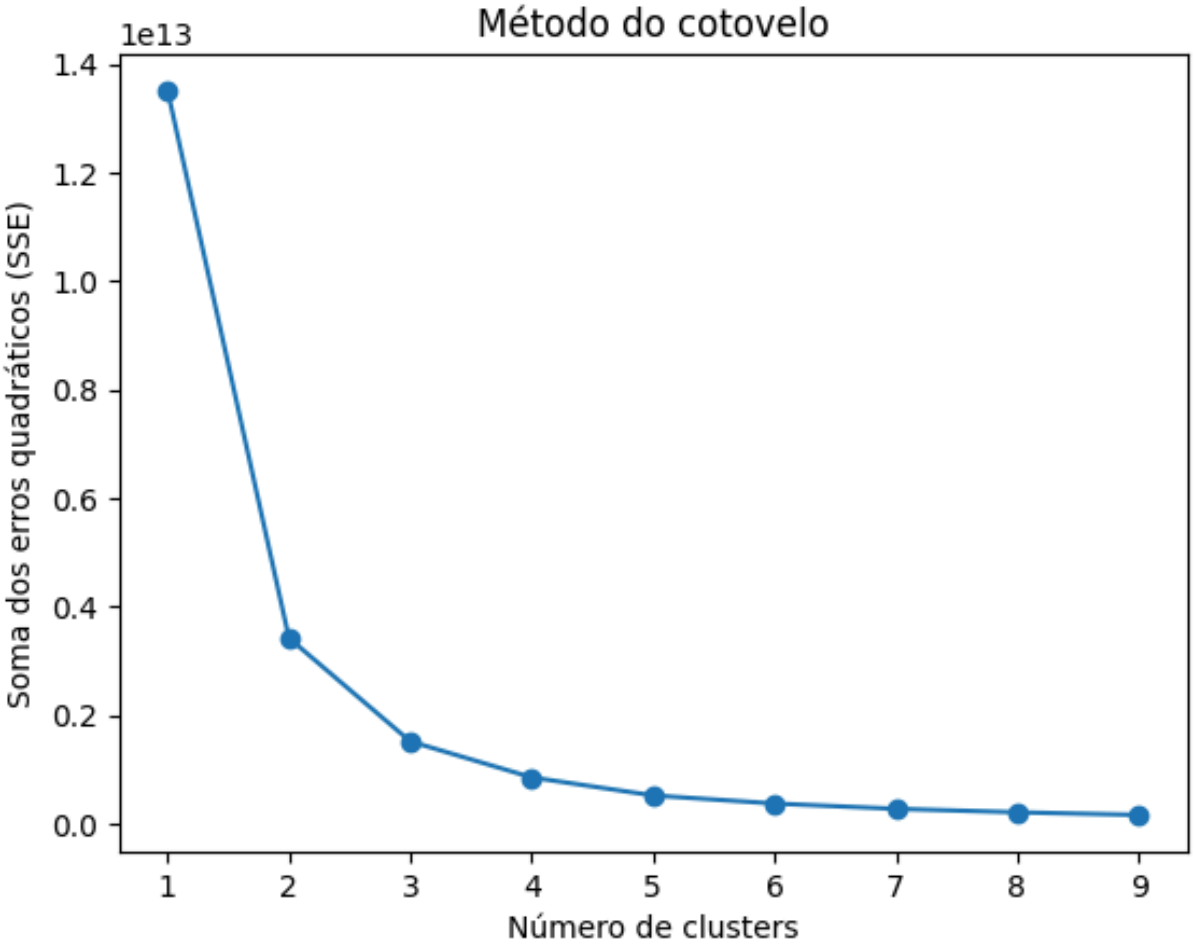

Figura 9.1: Resultado do método do cotovelo.

Dessa forma, é possível constatar que o valor crítico do `sse` está concentrado em apenas 2 colunas. No entanto, é válido questionar esse resultado, argumentando que, mesmo com 3 colunas, há uma diminuição no valor do `sse` como observado. Você tem razão neste ponto; no entanto, a diferença entre a redução obtida ao passar de 2 para 3 colunas é relativamente pequena. Portanto, para este caso, optaremos por manter apenas 2 colunas.

9.2 Criando o nosso algoritmo PCA

Como deduzimos que é possível reduzir o número de colunas do nosso conjunto, vamos criar o algoritmo PCA, mas antes vamos entender um pouco melhor como o PCA funciona.

Vimos que o PCA é uma técnica utilizada em análise de dados para reduzir a dimensionalidade de um conjunto de dados. Em outras palavras, ele permite a simplificação de um conjunto de dados que possui muitas colunas em um novo conjunto de dados com um número menor de colunas.

Para isso, o PCA identifica as correlações entre as variáveis presentes no conjunto de dados original. Ele utiliza essa informação para criar novas variáveis, chamadas de *componentes principais*, que são combinações lineares das variáveis originais. As novas variáveis são selecionadas de tal forma que capturem a maior quantidade possível de variação presente no conjunto de dados original. Essas novas variáveis são linearmente independentes, ou seja, não podem ser escritas como uma combinação linear das outras colunas e são ordenadas de acordo com a quantidade de variação que elas explicam no conjunto de dados original.

Ao utilizarmos o PCA para reduzir o número de colunas de um conjunto de dados, o objetivo é manter as novas variáveis mais importantes e descartar as menos importantes. Dessa forma, vamos simplificar o conjunto de dados sem perder informações consideradas importantes. Com esse objetivo em mente, podemos partir para a construção do algoritmo. Para isso, utilizaremos o `sklearn` e importaremos dele o PCA com o código a seguir:

```
from sklearn.decomposition import PCA
```

Em seguida, vamos inicializar o PCA e ajustá-lo ao nosso conjunto de dados:

```
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
```

Veja que o número de componentes (`n_components`) aqui é `2` ; isso acontece, pois já averiguamos antes qual seria o melhor valor. Em seguida, ajustamos e transformamos os nossos dados com o algoritmo PCA utilizando o `fit_transform` .

Por fim, vamos calcular o quanto dos nossos dados consegue ser explicado por apenas 2 componentes do nosso dataframe:

```
explicacao = sum(pca.explained_variance_ratio_)
print(f"Variância explicada pelos 2 componentes selecionados:
{explicacao}")
```

O resultado obtido com apenas 2 colunas foi de 0.99; ou seja, com 2 colunas, conseguimos explicar 99% dos nossos dados. Agora você pode estar se perguntando: mas quais colunas? Para descobrirmos quais as colunas que explicam melhor o nosso modelo, podemos utilizar o método `components_` da seguinte forma:

```
colunas = exames.columns[:-1]
pesos = pd.DataFrame(pca.components_, columns=colunas).iloc[0]
pesos = pesos.abs().sort_values(ascending=False)
top_5 = pesos[:5]
print("Top 5 colunas mais importantes:")
for coluna, importancia in zip(top_5.index, top_5.values):
    print(f"{coluna}: {importancia}")
```

Como resultado, temos a seguinte lista:

```
Top 5 colunas mais importantes:
plaquetas: 0.9999999917393074
leucocitos: 0.00012753004924126454
colesterol: 1.4667281109070375e-05
glicose: 4.3853331269227275e-06
ureia: 4.100134097382549e-06
```

As duas colunas selecionadas pelo método PCA são: `plaquetas` e `leucocitos` . Temos quase a totalidade dos nossos dados explicados com apenas essas duas colunas.

Embora esse tenha sido um exemplo simples com um dataframe inicial de 10 colunas, foi possível reduzi-lo para apenas 2 colunas sem perder muita informação, selecionando as colunas mais relevantes. Esse método de redução de dimensionalidade pode ser especialmente útil na área da saúde, pois permite reduzir o tempo de espera no exame e a quantidade de sangue a ser coletada para análise, ao mesmo tempo em que ajuda profissionais a identificarem quais variáveis são mais importantes para focar em suas análises.

9.3 Dados desbalanceados

Dados desbalanceados são comuns em muitos conjuntos de dados do mundo real. Esse desequilíbrio ocorre quando uma classe minoritária tem muito menos exemplos do que a classe majoritária. Infelizmente, os dados desbalanceados podem ser um grande desafio para a pessoa analista de dados. Por exemplo, um modelo de aprendizado de máquina que é treinado em um conjunto de dados desbalanceado pode ter uma tendência a classificar a maioria das amostras como pertencentes à classe majoritária, ignorando completamente outra classe.

Para lidar com esse problema, uma das soluções é aplicar técnicas de reamostragem, conhecidas como **upsampling** e **downsampling**. O *upsampling* envolve a duplicação de exemplos da classe minoritária para torná-la mais representativa no conjunto de dados. Já o *downsampling* envolve a redução do número de exemplos da classe majoritária para equilibrar a distribuição de classe no conjunto de dados.

Nesta seção veremos duas técnicas uma para aumento dos dados conhecida como *SMOOTE* e outra para redução dos dados conhecida como *Random Under Sampler*. Faremos a codificiação e discutiremos suas vantagens e desvantagens para lidarmos com conjuntos desbalanceados.

Começando com nossos dados

Para esta seção, continuaremos utilizando os dados que utilizamos no PCA, porém desta vez daremos ênfase à coluna `diagnostico`.

Para isso, vamos contar a incidência de cada doença em nossa tabela com o código a seguir:

```
contagem_doencas = exames['diagnostico'].value_counts().reset_index()
contagem_doencas.columns = ['diagnostico', 'contagem']
```

O resultado dessa contagem é o seguinte dataframe:

diagnostico	contagem
Hipercolesterolemia	462
Disfunção Renal	256
Insuficiência Renal	249
Diabetes	136
Deficiência de Vitamina D	125
Hipercalemia	114
Infecção	33
Anemia	23
Trombocitopenia	16

Observe que nosso dataframe contém diagnósticos com diferentes incidências, algumas mais comuns, outras menos. Essa desigualdade na quantidade de amostras de cada classe pode gerar um viés no algoritmo de *machine learning*, pois ele pode ser tendencioso e classificar incorretamente as doenças raras, que possuem menos amostras. Além disso, o modelo corre o risco de ser superajustado às doenças mais comuns, prejudicando a acurácia na classificação das doenças menos frequentes.

9.4 Gerando dados de minorias com SMOTE

Quando lidamos com dados desbalanceados, em que a classe minoritária possui poucas amostras em relação à classe majoritária, técnicas de geração de novos dados podem ser utilizadas para melhorar o desempenho de nosso modelo de aprendizado de máquina. A técnica que veremos é chamada de SMOTE (*Synthetic*

Minority Over-sampling Technique), disponível dentro da biblioteca chamada `imbalanced-learn`.

O SMOTE funciona gerando novas amostras sintéticas de classe minoritária. Para isso, ele seleciona aleatoriamente um ponto da classe e calcula os `k` vizinhos mais próximos desse ponto. Em seguida, ele seleciona aleatoriamente um dos vizinhos e gera um novo ponto sintético na reta que conecta os dois pontos. Esse processo é repetido até que o número desejado de novas amostras sintéticas seja gerado.

Entre as vantagens do SMOTE está a capacidade de gerar novas amostras que assemelham a distribuição dos dados originais da classe minoritária, ajudando a evitar *overfitting*. Além disso, ele é simples de implementar.

Entretanto, o SMOTE possui algumas desvantagens, como a possibilidade de gerar amostras que são muito semelhantes umas às outras, levando a uma superestimação da precisão do modelo. E ele pode ser sensível a valores discrepantes (os *outliers*) e pode aumentar significativamente o tamanho do conjunto de dados.

Antes de criarmos o código para o método SMOTE, deveremos criar um outro código para fazermos a codificação de valores categóricos para modificarmos os valores da coluna `diagnostico` para valores numéricos. Utilizaremos a codificação One-Hot como no código a seguir:

```
from sklearn.preprocessing import OneHotEncoder

X = exames.drop("diagnostico", axis=1).values
y = exames["diagnostico"].values

encoder = OneHotEncoder(sparse=False)
y_encoded = encoder.fit_transform(y.reshape(-1, 1))
```

No código que acabamos de executar fizemos a codificação One-Hot na coluna `diagnostico` para deixar em conformidade com o próximo

algoritmo, que será o algoritmo de SMOTE.

Vamos ao código para gerar mais dados para as doenças com menor número de incidências; para isso, vamos usar o método `SMOTE` da biblioteca `imblearn.over_sampling`, como a seguir:

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y_encoded)

y_resampled_decoded = encoder.inverse_transform(y_resampled)

exames_resampled = pd.DataFrame(X_resampled, columns=exames.columns[:-1])
exames_resampled["Doenca"] = y_resampled_decoded
```

Vamos entender o que fizemos nesse código:

1. Começamos importando a biblioteca que contém o método que utilizaremos para criar o `SMOTE`;
2. Instanciamos o método `SMOTE`;
3. Declaramos dois dataframes chamados `X_resampled` e `y_resampled` e, na mesma linha, aplicamos o método `fit_resample` para realizar o *oversampling* dos dados onde `X` são os dados de entrada e `y_encoded` é a variável-alvo codificada do código `OneHotEncoder`;
4. Utilizamos o método `inverse_transform` do objeto `OneHotEncoder` instanciado do código anterior para decodificar a matriz `y_resampled` de volta para suas representações originais de classe;
5. Criamos um novo dataframe para armazenar o nome das doenças;
6. Adicionamos nesse novo dataframe a contagem de cada uma das doenças.

Por fim, poderemos criar um código para analisarmos novamente a contagem de cada uma das doenças:

```
contagem_doencas_m =
exames_resampled["diagnostico"].value_counts().reset_index()
contagem_doencas.columns = ["diagnostico", "contagem"]
```

O resultado pode ser visto na tabela a seguir:

diagnostico	Contagem
Diabetes	462
Hipercalemia	462
Disfunção Renal	462
Insuficiência Renal	462
Hipercolesterolemia	462
Deficiência de Vitamina D	462
Anemia	462
Trombocitopenia	462
Infecção	462

A criação dos dados sintéticos da categoria rara foi concluída com êxito. Perceba que o algoritmo SMOTE aumentou todas as ocorrências das outras doenças para o mesmo número da doença "Hipercolesterolemia", que é a mais comum; porém, como efeito colateral, nosso dataframe aumentou de 1414 linhas para 4158 linhas. Em um dataframe pequeno como esse talvez não tenhamos problemas com desempenho, porém, à medida que aumentamos nossos conjuntos de dados, podemos ter problemas.

9.5 Reduzindo os dados da maioria

Em algumas situações, podemos ter que, ao invés de aumentar os dados da minoria, reduzi-los para não aumentarmos demais o nosso conjunto de dados. Nessas ocasiões, podemos utilizar o método com *downsampling*. A ideia do *downsampling* é reduzir a quantidade de amostras da classe majoritária para equilibrarmos a distribuição de classes.

A técnica que veremos para fazer o *downsampling* é conhecida como `RandomUnderSampler` da biblioteca `imbalanced-learn`. O `RandomUnderSampler` seleciona aleatoriamente uma amostra da classe majoritária para ser mantida no conjunto de dados final. Isso é feito até que o número de amostras na classe majoritária corresponda ao número de amostras na classe minoritária.

A principal vantagem dessa técnica é que ela pode ser mais rápida e menos propensa ao *overfitting* do que o *upsampling*. Além disso, como já mencionamos, o *downsampling* evita que nosso conjunto de dados aumente e prejudique o tempo de treino do nosso modelo. Entretanto, a realização do *downsampling* pode descartar informações importantes do nosso conjunto de dados.

Outro fato que devemos analisar é que a questão de uma classe ser majoritária é algo subjetivo, ou seja, não existe um valor específico que a torne majoritária. O que vai dizer é a quantidade de dados que possuímos. Então, mesmo que uma classe seja considerada majoritária, podemos ter poucos dados a respeito dela em nosso dataframe e, ao utilizarmos o *downsampling*, vamos reduzir ainda mais sua quantidade em nosso dataframe.

Vamos criar o código para executar o `RandomUnderSampler` com os dados gerados pela codificação `OneHotEncoder` do código anterior:

```
from imblearn.under_sampling import RandomUnderSampler

rus = RandomUnderSampler(random_state=42)
X_resampled, y_resampled = rus.fit_resample(X, y_encoded)
```

```
y_resampled_decoded = encoder.inverse_transform(y_resampled)

exames_resampled = pd.DataFrame(X_resampled, columns=exames.columns[:-1])
exames_resampled["Diagnostico"] = y_resampled_decoded
```

Vamos entender o que fizemos nesse código:

1. Na primeira linha, importamos o método `RandomUnderSampler` da biblioteca `imblearn.under_sampling`;
2. Em seguida, instanciamos nosso método na variável `rus`;
3. Ajustamos nossos dados com o método `rus.fit` passando como parâmetros o dataframe `X` e o `y_encoded`;
4. Depois de feito o **downsampling**, utilizamos o método `encoder.inverse` do método `OneHotEncoder` para voltarmos a variável *y* à sua codificação original, ou seja, com o nome dos diagnósticos. Para finalizar podemos fazer a contagem de nossos diagnósticos com o seguinte código:

```
contagem_doencas_d =
exames_resampled["diagnostico"].value_counts().reset_index()
contagem_doencas.columns = ["diagnostico", "contagem"]
contagem_doencas_d.head()
```

O resultado pode ser visto na tabela a seguir:

diagnostico	contagem
Anemia	16
Deficiência de Vitamina D	16
Diabetes	16
Disfunção Renal	16
Hipercalemia	16
Hipercolesterolemia	16
Infecção	16
Insuficiência Renal	16
Trombocitopenia	16

Com isso, nós reduzimos o nosso dataframe que antes tinha 1414 linhas para apenas 144 linhas, o que é algo bem significativo, mas pode ser prejudicial para nossos algoritmos de *machine learning*. Ao escolhermos entre os métodos de *upsampling* e *downsampling*, é importante considerar as características específicas do conjunto de dados em questão e o objetivo do projeto. Ambas as técnicas têm suas vantagens e desvantagens.

O *upsampling* é geralmente útil quando o conjunto de dados possui poucas amostras da classe minoritária, e aumentar a quantidade de amostras pode melhorar o desempenho dos algoritmos de *machine learning*. Porém, é preciso ter cuidado para que não criemos amostras sintéticas que sejam semelhantes às já existentes e que acabem levando ao *overfitting* do modelo.

O *downsampling* é geralmente útil quando o conjunto de dados possui uma quantidade desproporcional de amostras de classe majoritária e o objetivo é equilibrar as classes. Entretanto, seu uso pode levar à perda de informações importantes e diminuir a capacidade do modelo de generalizar.

No próximo capítulo veremos mais sobre as técnicas de redução e aumento do nosso dataframe com as técnicas de *upsampling* e *downsampling*.

Capítulo 10
Projeto prático — Redução de dimensionalidade com upsampling e downsampling

Neste projeto prático vamos observar o *upsampling* e *downsampling* e veremos como essas técnicas podem afetar os algoritmos de *machine learning*. Para isso, utilizaremos um conjunto de dados sintéticos:

paciente	idade	sexo	medicamento	dosagem	p
1	69	Masculino	Tipo_B	171	1
2	32	Masculino	Tipo_B	79	1
3	89	Feminino	Tipo_B	237	1
4	78	Masculino	Tipo_A	413	1
5	38	Masculino	Tipo_B	403	1

pressaopos	colesterolpre	colesterolpos	efeitoadverso
109	276	160	Nenhum
124	282	204	Nenhum
121	197	200	Nenhum
137	152	195	Nenhum
133	265	199	Nenhum

A seguir, a explicação dos significados de cada coluna:

- paciente: Identificador único para cada paciente, de 1 a 1000.
- idade: Idade do paciente, variando entre 18 e 90 anos.
- sexo: Sexo do paciente, sendo `Masculino` ou `Feminino`.
- medicamento: Tipo de medicamento administrado (`Tipo_A` , `Tipo_B` , `Tipo_C`) com diferentes probabilidades de uso.
- dosagem: Dosagem do medicamento administrado, variando de 50 mg a 500 mg.
- pressaopre: Pressão arterial pré-tratamento.
- pressaopos: Pressão arterial pós-tratamento.
- colesterolpre: Nível de colesterol pré-tratamento.
- colesterolpos: Nível de colesterol pós-tratamento.
- efeitoadverso: Efeitos adversos do medicamento, com categorias `Nenhum` , `Leve` , e `Grave` .
- resultado: Resultado do tratamento, podendo ser `Positivo` ou `Negativo` .

Agora, que já conhecemos nosso dataframe vamos fazer a análise exploratória de dados.

10.1 Análise exploratória de dados

A análise exploratória de dados é um passo fundamental em qualquer projeto de Ciência de Dados e engenharia de recursos, pois é a partir dela que conseguimos compreender a estrutura dos nossos dados e identificar possíveis padrões e caracteristicas essenciais para a melhor compreensão do nosso problema. Então vamos realizar essa análise exploratória com a intenção de encontrar aspectos importantes como desbalanceamento dos dados para justificar o uso de técnicas como *upsampling* e o *downsampling* dos nossos dados.

O primeiro passo é examinar as características gerais dos dados, incluindo estatísticas descritivas e a distribuição das variáveis. Para isso utilizaremos o método `describe` :

```
df.describe()
```

index	paciente	idade	dosagem	pressaopre
count	1000.0	1000.0	1000.0	1000.0
mean	500.5	53.33	281.10	150.77
std	288.81	21.15	131.47	17.60
min	1.0	18.0	50.0	120.0
25%	250.75	35.0	165.75	136.0
50%	500.5	53.0	285.0	151.0
75%	750.25	72.0	395.25	166.0
max	1000.0	90.0	500.0	180.0

pressaopos	colesterolpre	colesterolpos
1000.0	1000.0	1000.0
120.03	224.86	179.92
11.44	43.20	35.0
100.0	150.0	120.0
111.0	189.0	149.0
120.0	224.0	180.0
130.0	262.0	209.0
140.0	300.0	240.0

Após esse código vamos fazer a contagem de cada uma das colunas da seguinte forma:

```
colunas = ["sexo","medicamento","efeito","resultado]

for coluna in colunas:
    print(f"Contagem de categorias da coluna: {coluna}\n
{df[coluna].value_counts()}")
```

O resultado pode ser visto a seguir:

```
Contagem de categorias da coluna: sexo
sexo
Feminino     510
Masculino    490
Name: count, dtype: int64

Contagem de categorias da coluna: medicamento
medicamento
Tipo_A    482
Tipo_B    320
Tipo_C    198
Name: count, dtype: int64
```

```
Contagem de categorias da coluna: efeitoadverso
efeitoadverso
Nenhum    849
Leve      115
Grave      36
Name: count, dtype: int64

Contagem de categorias da coluna: resultado
resultado
Positivo    920
Negativo     80
Name: count, dtype: int64
```

Conclusões:

A análise de contagem das categorias nas colunas `sexo`, `medicamento`, `efeitoadverso` e `resultado` nos fornece os seguintes resultados:

1. Distribuição por sexo: a coluna `sexo` está bem equilibrada, com 503 pacientes do sexo masculino e 497 do sexo feminino. Isso indica que o conjunto de dados não apresenta viés significativo em relação ao sexo dos pacientes, o que é importante para garantir que qualquer análise ou modelagem subsequente não seja influenciada por uma desproporção entre os sexos.

2. Distribuição por tipo de medicamento: a coluna `medicamento` mostra que o `Tipo_A` é o mais administrado, com 503 pacientes, seguido pelo `Tipo_B`, com 308 pacientes, e `Tipo_C`, que foi administrado a 189 pacientes. Esse desequilíbrio pode refletir a preferência ou eficácia percebida do `Tipo_A` em relação aos outros medicamentos, ou pode ser um reflexo da prescrição mais frequente desse medicamento. Devemos considerar essa distribuição em análises futuras para garantir que os resultados não sejam enviesados por uma prevalência de um tipo específico de medicamento.

3. Distribuição de efeitos adversos: a maioria dos pacientes, 848, não apresentou efeitos adversos, enquanto 109 relataram efeitos leves e 43 tiveram efeitos graves. Essa distribuição indica que, na maioria dos casos, os medicamentos são bem tolerados, mas uma parcela significativa ainda experimenta algum tipo de efeito adverso.

4. Distribuição dos resultados do tratamento: a coluna `resultado` revela que 906 pacientes tiveram um resultado positivo ao tratamento, enquanto apenas 94 tiveram um resultado negativo. Esse desbalanceamento em nosso conjunto de dados sugere que o tratamento é eficaz. Entretanto, o número baixo dos casos negativos pode dificultar a modelagem preditiva para essa classe, tornando necessária a aplicação de técnicas como *upsampling* ou *downsampling* em nosso conjunto dados.

O próximo passo é a visualização e a distribuição das variáveis numéricas vai nos ajudar a identificar possíveis outliers e entender a variabilidade dos dados, para isso utilizaremos o código a seguir:

```
df[["idade","dosagem","pressaopre","pressaopos","colesterolpre","colesterolpos"]].hist(bins=20,figsize=(12,10))
plt.show()
```

O resultado pode ser visto na imagem a seguir:

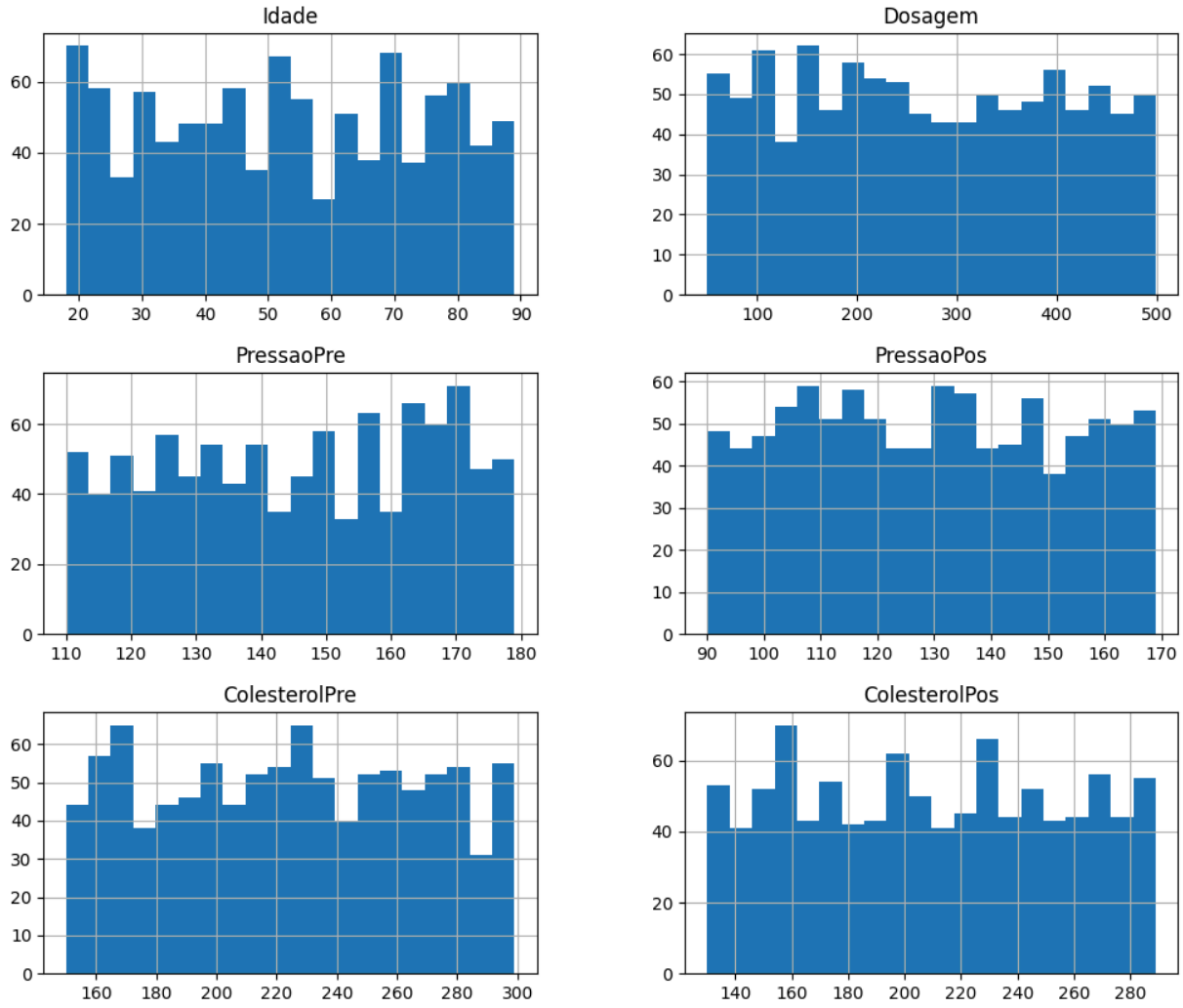

Figura 10.1: Distribuição das variáveis numéricas

Os histogramas apresentados refletem a distribuição das variáveis em nosso conjunto de dados. Os gráficos mostram a distribuição das seguintes variáveis: `idade`, `dosagem`, `pressaopre`, `pressaopos`, `colesterolpre`, `colesterolpos`. A partir da visualização dos histogramas, podemos tirar algumas conclusões:

1. `idade`: a distribuição das idades parece ser aproximadamente uniforme, com uma ligeira concentração entre 10 e 30 anos. Isso indica que há uma distribuição relativamente balanceada das idades no conjunto de dados, com um destaque nas faixas etárias consideradas mais jovens.

2. `dosagem` : a variável dosagem apresenta uma distribuição também relativamente uniforme, com algumas flutuações nas frequências. Isso pode sugerir que diferentes dosagens foram aplicadas de maneira equilibrada, sem uma dosagem claramente dominante.

3. `pressaopre` e `pressaopos` : ambas as variáveis de pressão exibem distribuições que parecem seguir um padrão semelhante, com pequenas variações ao longo dos intervalos de pressão. Podemos observar uma tendência de maior concentração em faixas intermediárias (110 e 140 na coluna `pressaopre` e 110 a 130 na coluna `pressaopos`), sugerindo a nós que a maioria dos pacientes tinha pressão arterial dentro desses intervalos, tanto antes quanto depois do tratamento.

4. `colesterolpre` e `colesterolpos` : as distribuições das variáveis relacionadas ao colesterol (pré e pós) também são uniformes, mas com algumas variações. Conseguimos observar uma distribuição relativamente equilibrada entre as faixas de valores de colesterol, com uma leve concentração em faixas intermediárias (160 a 280).

Relação entre variáveis

Ao explorarmos as relações entre as diferentes variáveis podemos encontrar possíveis correlações e interações importantes, especialmente entre variáveis que podem impactar o resultado do tratamento:

```
sns.pairplot(df, vars=['pressaopre', 'pressaopos', 'colesterolpre',
'colesterolpos'], hue='resultado')
plt.show()
```

O resultado pode ser visto a seguir:

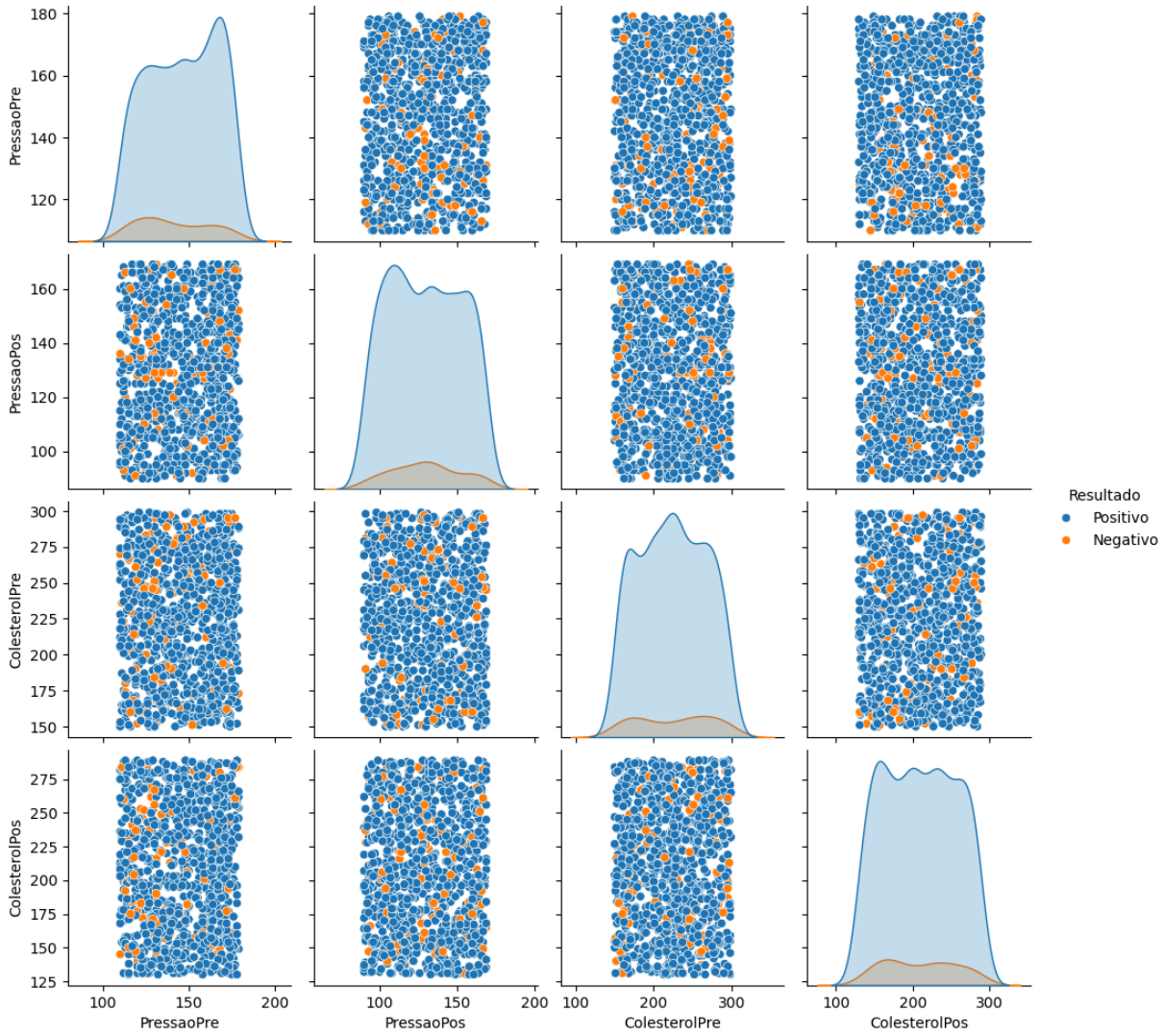

Figura 10.2: Relação entre as variáveis

A imagem que geramos mostra a relação entre as variáveis `pressaopre`, `pressaopos`, `colesterolpre`, `colesterolpos`, colorindo os dados de acordo com a variável categórica `resultado`, que tem os valores `Positivo` e `Negativo`.

Conclusões

1. Distribuição univariada: na diagonal do gráfico, temos as distribuições univariadas das variáveis individuais. Todas as variáveis apresentam distribuições similares, com a maioria dos dados concentrados em intervalos específicos. Notamos uma

concentração significativa de valores em faixas específicas, como em torno de 120-160 para a pressão arterial e 160-280 para o colesterol, o que é consistente com os histogramas observados anteriormente.

2. Dispersões e relações bivariadas: no restante dos resultados, observamos gráficos de dispersão que mostra como as variáveis se relacionam entre si.

No geral, há uma distribuição dispersa, com poucas relações visíveis entre as variáveis. Os pontos de `Positivo` e `Negativo` estão misturados de forma bastante aleatória, sem uma clara separação entre eles. Isso pode indicar que as variáveis de pressão e colesterol, isoladamente, não são bons preditores do resultado.

Em seguida veremos a matriz de correlação destes dados:

```
correlation_matrix = df.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.show()
```

A matriz de correlação pode ser vista a seguir:

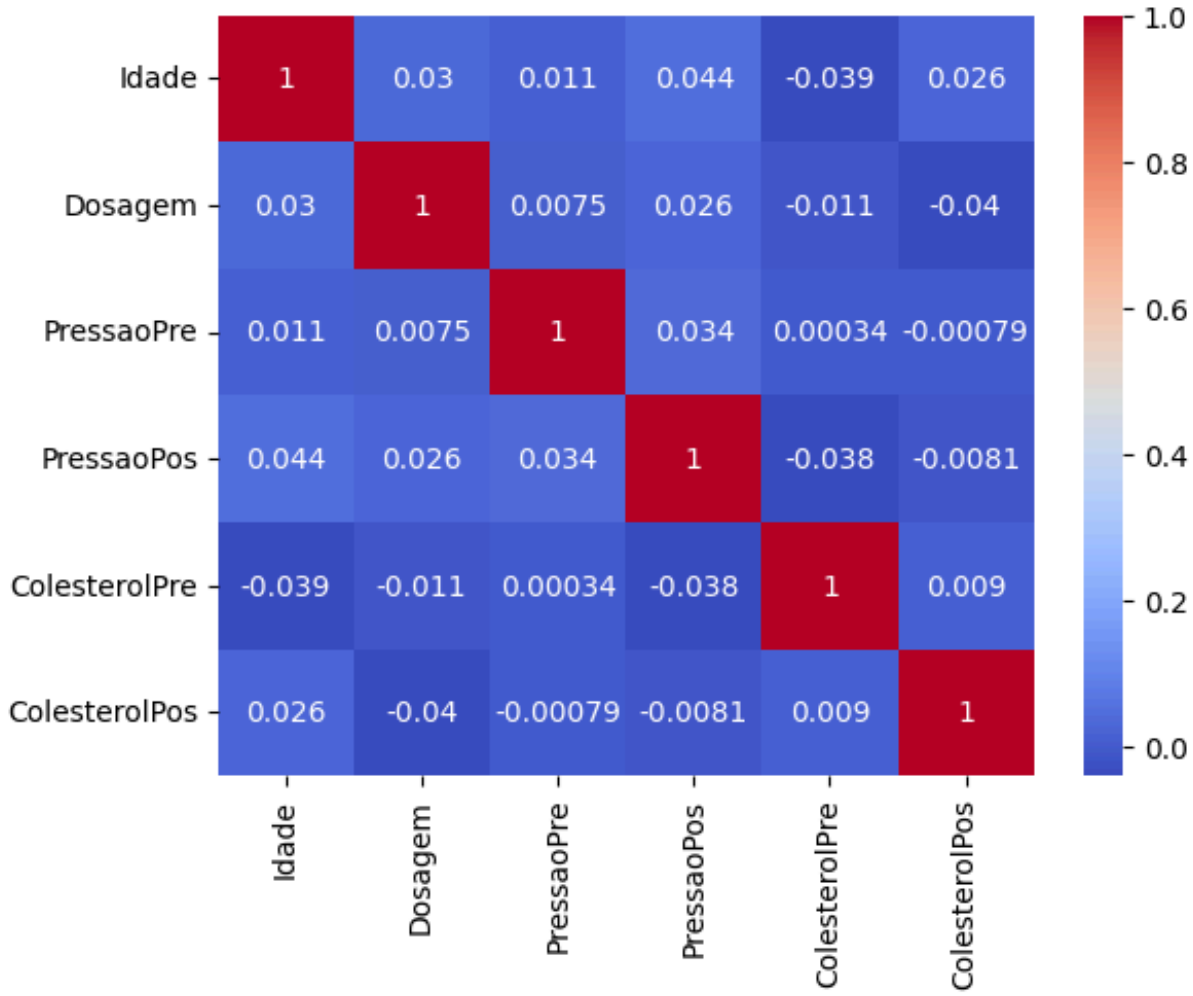

Figura 10.3: Matriz de correlação entre as variáveis

Conclusões:

Com a matriz de correlação feita conseguimos chegar às seguintes conclusões:

1. Correlações identificadas

- **Idade e as demais variáveis**: Existe uma correlação muito fraca entre a idade dos pacientes e variáveis como pressão arterial, colesterol ou dosagem do medicamento. Isso sugere que a idade, isoladamente, não influencia diretamente essas variáveis.

- **Doagem e demais variáveis**: A dosagem do medicamento também apresenta correlação muito baixa com todas as variáveis, indicando que sua variação é praticamente independente das demais.
- **Pressão pré e pós-tratamento**: Embora a correlação entre `pressaopre` e `pressaopos` seja mais significativa que outras variáveis, ainda é baixa, sugerindo que o tratamento pode ter alterado a pressão arterial de forma não linear ou sob influência de outros fatores.
- **Colesterol pré e pós-tratamento**: Os níveis de colesterol antes e após o tratamento apresentam uma relação levemente positiva, indicando que há alguma consistência entre os valores, mas sem uma correlação forte

2. Interpretação geral

- O gráfico revela que, no conjunto de ados analisado, as variáves escolhidas não têm relações lineares fortes entre si
- Essa baixa correlação sugere que outros fatores ou interações não representados na matriz podem estar influenciado os resultados

Análise de classes desbalanceadas

Para técnicas de *upsampling* e *dowsampling*, precisamos entender como as classes estão distribuídas, para sabermos ou não se precisaremos utilizá-los. Em nosso caso em específico analisaremos as classes `efeitoadverso` e `Resultado` :

```
sns.countplot(x='efeitoadverso', data =df)
plt.title("Distribuição de Efeito Adverso')
plt.show()
```

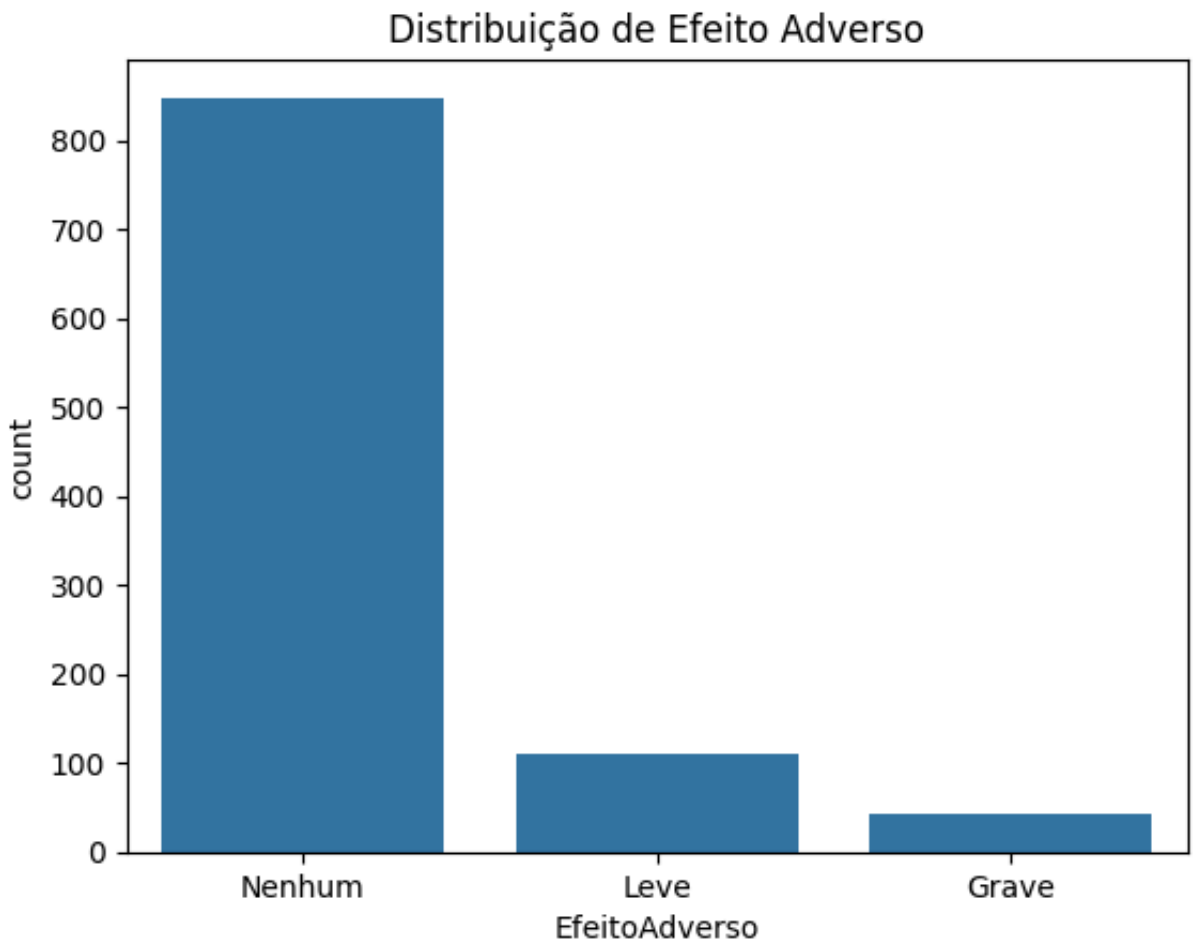

Figura 10.4: Distribuição do efeito adverso

Neste primeiro gráfico conseguimos observar a distribuição de efeito adverso, em que temos uma maior concentração da classe `Nenhum` seguido de `Leve` e `Grave`. A menor representatividade das classes `Leve` e `Grave` pode gerar distorções na modelagem fazendo com que o algoritmo de *machine learning* não consiga extrair as características dessas duas classes.

Prosseguindo, vamos fazer o código a seguir:

```
sns.countplot(x='resultado',data=df)
plt.title("Distribuição do resultado do tratamento")
plt.show()
```

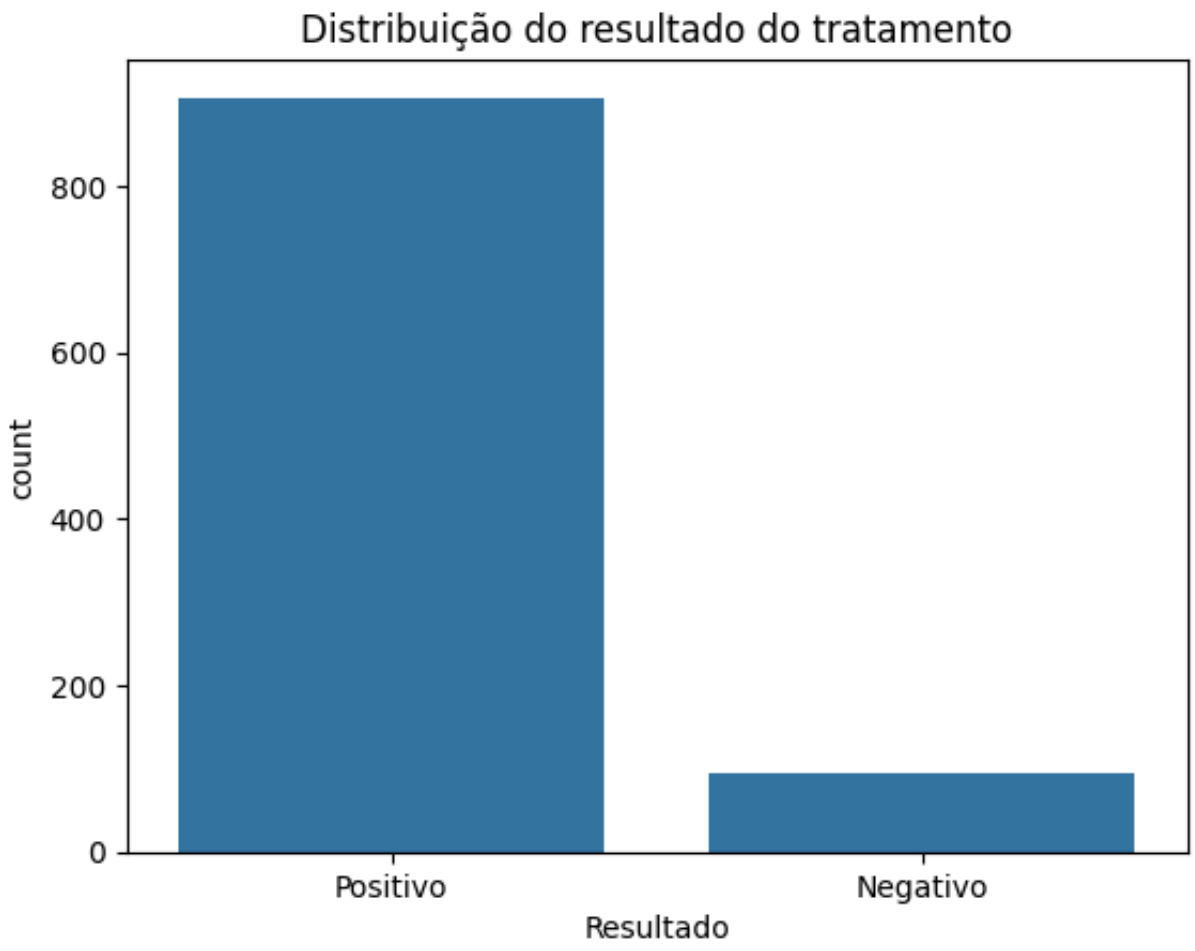

Figura 10.5: Distribuição do resultado

A classe `Positivo` é muito mais representativa aos nossos dados do que a classe `Negativo`, dessa forma, nosso algoritmo de *machine learning* pode ter dificuldade na extração de características da classe `Negativo` levando a previsões equivocadas.

A conclusão a que chegamos é que esse conjunto de dados apresenta um certo grau de desbalanceamento significativo entre as classes de resultados (Positivo e Negativo).

10.2 Utilizando técnicas de upsampling e downsampling

O desbalanceamento de classes é um problema em conjuntos de dados de saúde, pois pode nos levar a modelos preditivos que são enviesados em favor da classe majoritária. Em outras palavras, se a maioria dos exemplos no conjunto de dados pertence a uma classe específica, o modelo pode "aprender" a prever predominantemente essa classe, resultando em um desempenho menor para classe minoritária. Em nosso contexto específico, que é a área de saúde, isso pode ser especialmente perigoso, pois a classe minoritária pode, por exemplo, representar pacientes com uma condição grave que não deve ser ignorada ou subestimada pelo modelo.

Então, nesta seção, faremos tanto o *upsampling* quanto o *downsampling* para apresentar a acurácia nos dados de *machine learning*. Porém, antes, vamos rodar o modelo de *machine learning* com os dados desbalanceados. Dessa forma, conseguiremos observar as mudanças na precisão de dados após aplicarmos as técnicas de balanceamento.

O código que utilizaremos fará uma árvore de decisão para prever o resultado e, em seguida, calcularemos a acurácia deste modelo utilizando os dados desbalanceados:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.utils import resample

# Separando as variáveis dependentes e independentes
X = df.drop('Resultado', axis=1)
y = df['Resultado']

X = pd.get_dummies(X, drop_first=True)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Função para treinar e avaliar o modelo
def treinar_avaliar_modelo(X_train, y_train, X_test, y_test):
    modelo = RandomForestClassifier(random_state=42)
    modelo.fit(X_train, y_train)
    y_pred = modelo.predict(X_test)
    acuracia = accuracy_score(y_test, y_pred)
    return acuracia

acuracia_desbalanceado = treinar_avaliar_modelo(X_train, y_train, X_test,
y_test)
print(f"Acurácia com dados desbalanceados: {acuracia_desbalanceado:.4f}")
```

E o resultado da acurácia pode ser visto a seguir:

```
Acurácia com dados desbalanceados: 0.8967
```

Desempenho: a acurácia do modelo é alta, o que indica que ele está performando bem em termos gerais. Entretanto, isso pode ser enganoso em um conjunto de dados desbalanceado.

Viés de classe: como o conjunto de dados é desbalanceado, com a maioria significativa de resultados `Positivos`, o modelo provavelmente está inclinado a prever a classe majoritária. Isso significa que, embora a acurácia seja alta, o modelo pode não estar capturando corretamente a classe minoritária `Negativo`, que é crítica para prever falhas no tratamento.

Aplicaremos a técnica de *upsampling* para resolvermos o problema de desbalanceamento de classes no conjunto de dados de treinamento. Para isso, replicamos aleatoriamente as amostras da classe `Negativo` até igualá-las ao número de amostras da classe `Positivo`. Em seguida, combinamos essas amostras replicadas com as originais da classe `Positivo`, formando um novo conjunto de treinamento equilibrado.

Com o conjunto de dados balanceado, treinaremos o modelo e calcularemos a acurácia para avaliar o impacto do *upsampling* na performance.

```
X_train_up, y_train_up = resample(
        X_train[y_train == 'Negativo'],
        y_train[y_train == 'Negativo'],
        replace=True,
        n_samples=len(y_train[y_train == 'Positivo']),
    random_state=42)

X_train_up = pd.concat(
    [X_train[y_train == 'Positivo'], X_train_up])

y_train_up = pd.concat(
    [y_train[y_train == 'Positivo'], y_train_up])

acuracia_up = treinar_avaliar_modelo(
    X_train_up, y_train_up, X_test, y_test)

print(f"Acurácia com upsampling: {acuracia_up:.4f}")
```

O resultado pode ser visto a seguir:

```
Acurácia com upsampling: 0.8900
```

Desempenho: a acurácia é ligeiramente menor que com os dados desbalanceados, mas ainda assim é próxima.

Equilíbrio: o uso de *upsampling* ajudou a dar mais peso à classe minoritária `Negativo`, permitindo que o modelo capturasse melhor os casos que não estavam sendo previstos com os dados desbalanceados. Isso sugere um *trade-off* saudável, onde uma pequena perda na acurácia geral pode ter resultado em um modelo mais justo, especialmente na previsão da classe de maior interesse clínico.

E por fim vamos executar a árvore de decisão em um conjunto de dados com *downsampling*:

```
X_train_down, y_train_down = resample(
        X_train[y_train == 'Positivo'],
        y_train[y_train == 'Positivo'],
        replace=False,
        n_samples=len(y_train[y_train == 'Negativo']),
    random_state=42)

X_train_down = pd.concat(
    [X_train[y_train == 'Negativo'], X_train_dow])

y_train_down = pd.concat(
    [y_train[y_train == 'Negativo'], y_train_dow])

acuracia_down = treinar_avaliar_modelo(
    X_train_down, y_train_down, X_test, y_test)

print(f"Acurácia com downsampling: {acuracia_down:.4f}")
```

Podemos observar o resultado a seguir:

```
Acurácia com downsampling: 0.5800
```

Desempenho: a acurácia caiu significativamente. Isso é esperado, pois, ao reduzirmos a quantidade de dados da classe majoritária `Positivo`, o modelo tem menos informações para treinar, o que pode resultar em um desempenho inferior.

Efeito do *downsampling*: embora o *downsampling* tenha equilibrado o conjunto de dados, ele pode ter retirado informações importantes para o nosso modelo, levando à redução da nossa acurácia. Ou seja, esse método pode ser útil para conseguirmos retirar o viés dos nossos dados, porém o custo neste caso foi uma perda significativa da nossa acurácia.

Entretanto, a acurácia não é única métrica que podemos utilizar para analisarmos a precisão do nosso modelo de *machine learning*. Vamos utilizar a matriz de confusão para analisarmos a precisão do nosso modelo.

Matriz de confusão para os dados desbalanceados

A matriz de confusão ajuda a visualizar o desempenho do modelo, mostrando quantos exemplos de cada classe foram corretamente ou incorretamente classificados. Ao analisar a matriz de confusão para os dados desbalanceados, você pode demonstrar que, apesar da alta acurácia, o modelo pode estar ignorando a classe minoritária.

Então, faremos a matriz de confusão para os dados desbalanceados, para os dados com *upsampling* e com *downsampling*.

> **MATRIZ DE CONFUSÃO E MÉTRICAS**
>
> Para saber mais sobre a matriz de confusão e métricas como precisão, *recall* e F1-Score, confira uma explicação mais detalhada no apêndice.

```
from sklearn.metrics import confusion_matrix, classification_report,
roc_auc_score, roc_curve

modelo.fit(X_train, y_train)
y_pred_desbalanceado = modelo.predict(X_test)

def avaliar_modelo(y_test, y_pred, title):
    print(f"\n{title}")
    print("Matriz de Confusão:")
    cm = confusion_matrix(y_test, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(f'Matriz de Confusão - {title}')
    plt.xlabel('Predito')
    plt.ylabel('Verdadeiro')
    plt.show()

    print("\nRelatório de Classificação:")
    print(classification_report(y_test, y_pred))
```

```
avaliar_modelo(y_test, y_pred_desbalanceado, "Dados Desbalanceados")
```

A matriz pode ser vista na imagem a seguir:

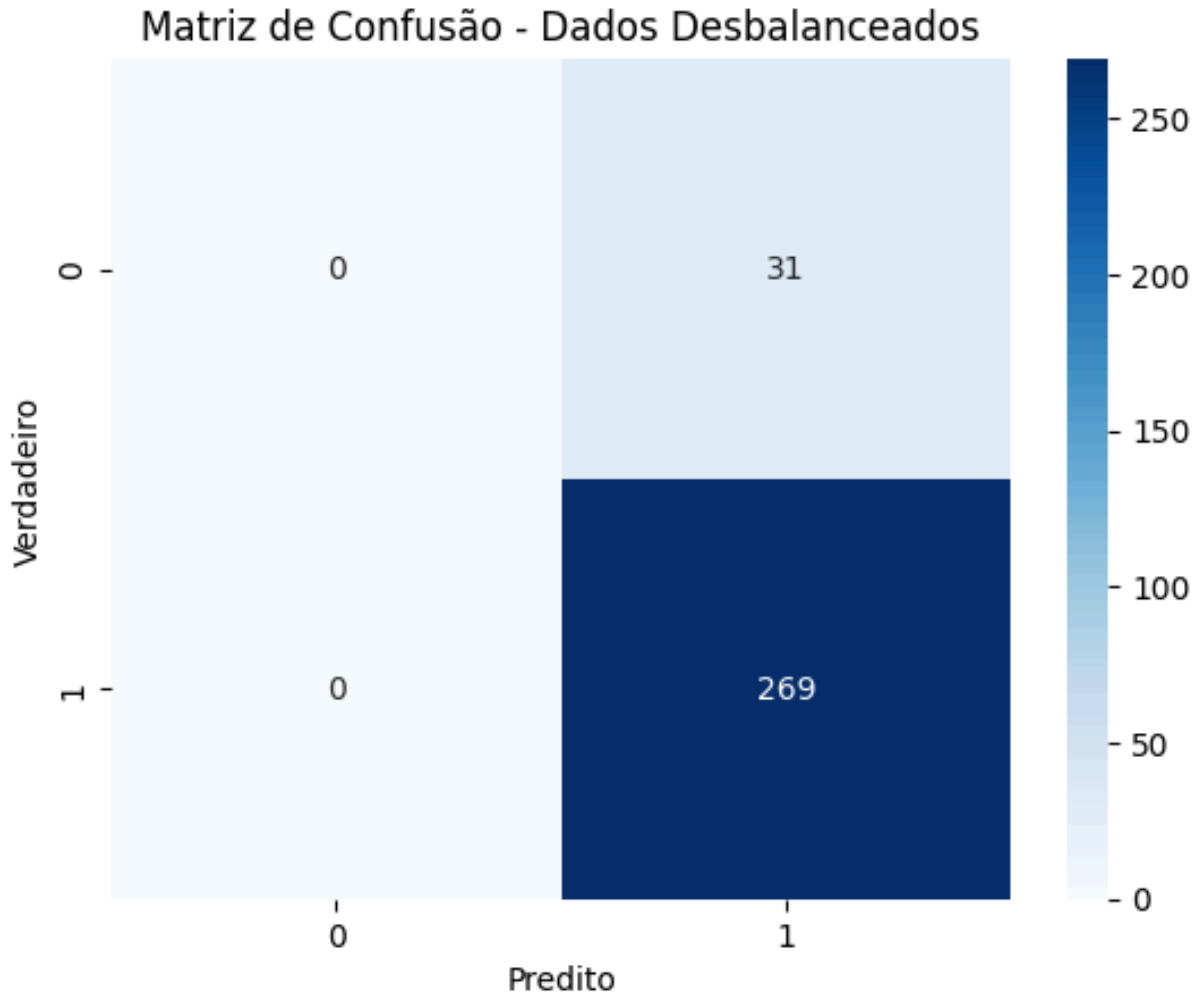

Figura 10.6: Matriz de confusão desbalanceada

E o relatório de classificação pode ser visto a seguir:

```
Relatório de Classificação:
              precision    recall  f1-score   support

    Negativo       0.00      0.00      0.00        31
    Positivo       0.90      1.00      0.95       269
```

```
    accuracy                           0.90       300
   macro avg       0.45      0.50      0.47       300
weighted avg       0.80      0.90      0.85       300
```

Vamos conferir o quão bom foi nosso modelo antes de termos feito os tratamentos de *upsampling* e *downsampling*. Isso vai nos ajudar a comparar a possível melhora que nosso tratamento de dados surtirá no desempenho do nosso modelo:

Precisão (precision)

- Negativo: 0.00 (o modelo não previu corretamente nenhum caso da classe `Negativo`).
- Positivo: 0.90 (o modelo é bom em prever a classe `Positivo`).

Revogação (Recall)

- Negativo: 0.00 (o modelo não conseguiu identificar casos da classe `Negativo`).
- Positivo: 1.00 (o modelo identificou corretamente todos os casos da classo `Positivo`).

F1-Score

- Negativo: 0.00 (a F1-Score para classe `Negativo` é 0.00, refletindo a incapacidade do modelo de prever corretamente a classe `Negativo`.
- Positivo: 0.95 (a F1-Score para classe `Positivo` permanece alta, indicando um bom desempenho na revisão dessa classe).

Acurácia: 0.90 (apesar da acurácia alta 0.90, o modelo praticamente ignora a classe `Negativo`, acertando apenas a classe `Positivo`. Esse comportamento é comum em modelos treinados com dados desbalanceados, onde o modelo aprende a priorizar a classe majoritária).

Agora, vamos fazer a matriz de confusão com os dados tratados com o *upsampling* usando o código a seguir:

```
modelo.fit(X_train_up, y_train_up)
y_pred_up = modelo.predict(X_test)
avaliar_modelo(y_test, y_pred_up, "Upsampling")
```

A matriz de confusão pode ser vista na imagem a seguir:

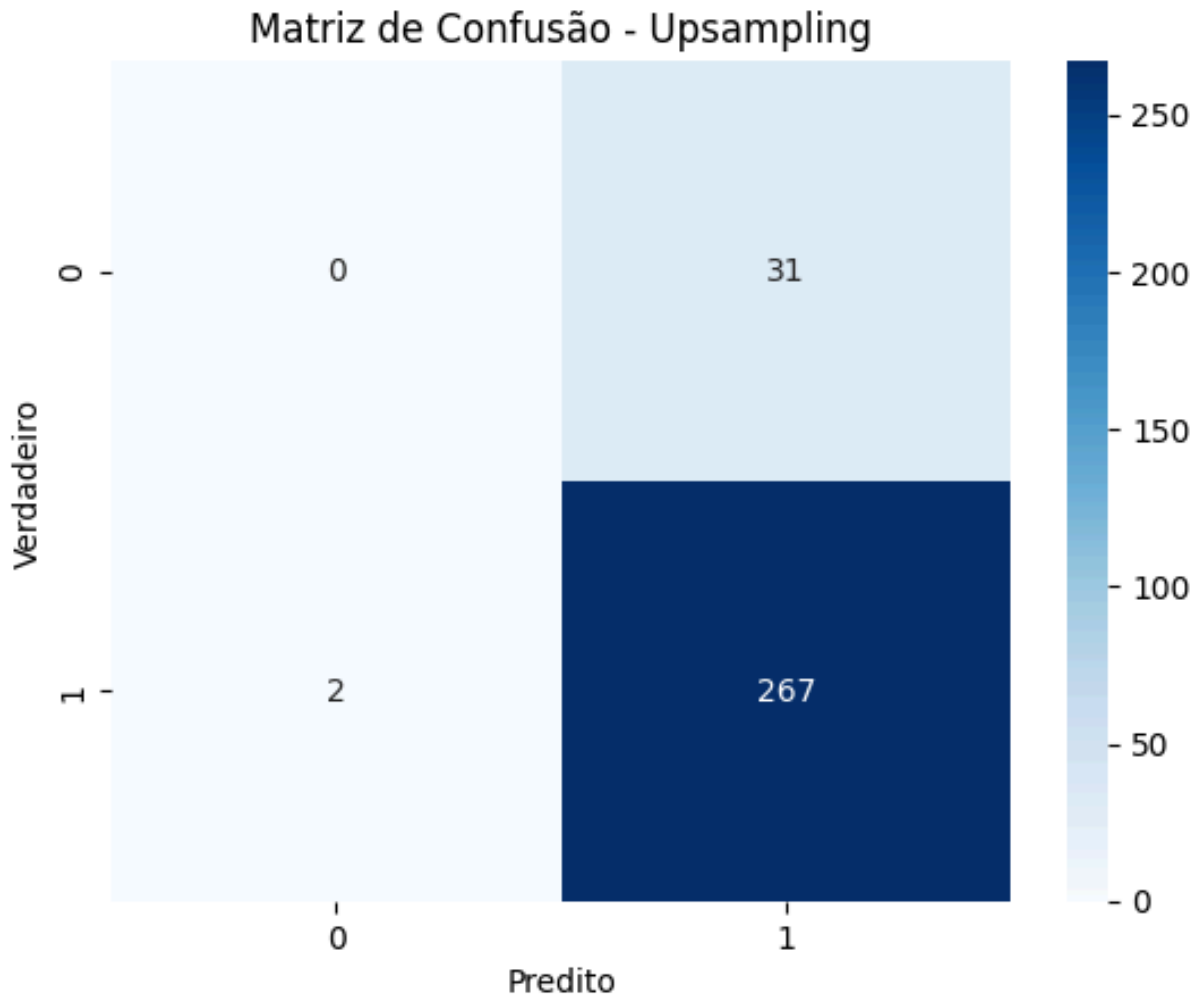

Figura 10.7: Matriz de confusão upsampling

E o relatório de classificação pode ser visto a seguir:

```
Relatório de Classificação:
              precision    recall  f1-score   support

    Negativo       0.00      0.00      0.00        31
    Positivo       0.90      0.99      0.94       269
```

```
    accuracy                           0.89       300
   macro avg       0.45      0.50      0.47       300
weighted avg       0.80      0.89      0.84       300
```

Vamos fazer a análise da nossa matriz de classificação com os dados tratados com *upsampling*:

Precisão (Precision)

- Negativo: 0.00 (o modelo não previu corretamente nenhum caso da classe `Negativo`, indicando que a classe `Negativo` ainda não está sendo capturada pelo modelo, mesmo com o *upsampling*).
- Positivo: 0.90 (o modelo continua a ser bom em prever a classe `Positivo`, com uma alta precisão).

Revogação (Recall)

- Negativo: 0.00 (o modelo não identificou nenhum caso da classe `Negativo`, apesar de o *upsampling* ter sido realizado, o que sugere que o problema com a classe minoritária persiste).
- Positivo: 0.99 (o modelo identificou corretamente quase todos os casos da classe `Positivo`, mostrando que a classe majoritária ainda está bem representada).

F1-Score

- Negativo: 0.00 (a F1-Score para classe `Negativo` é 0.00, refletindo a incapacidade do modelo de prever corretamente a classe `Negativo`).
- Positivo:0.94 (a F1-Score para classe `Positivo` permanece alta, indicando um bom desempenho na revisão dessa classe).

Acurácia: 0.89 (embora a acurácia continue alta 0.89, o modelo continua a ignorar a classe `Negativo`, acertando principalmente a classe `Positivo`. Isso evidencia que a abordagem de *upsampling* pode não ter sido suficiente para resolver o desbalanceamento de forma eficaz).

Por fim, vamos gerar a matriz de confusão com o nosso conjunto de dados tratados com o *downsampling* com o código a seguir:

```
modelo.fit(X_train_down, y_train_down)
y_pred_down = modelo.predict(X_test)
avaliar_modelo(y_test, y_pred_downmodelo.fit(X_train_downsampled,
y_train_downsampled)
y_pred_downsampled = modelo.predict(X_test), "Downsampling")
```

A matriz de confusão pode ser vista a seguir:

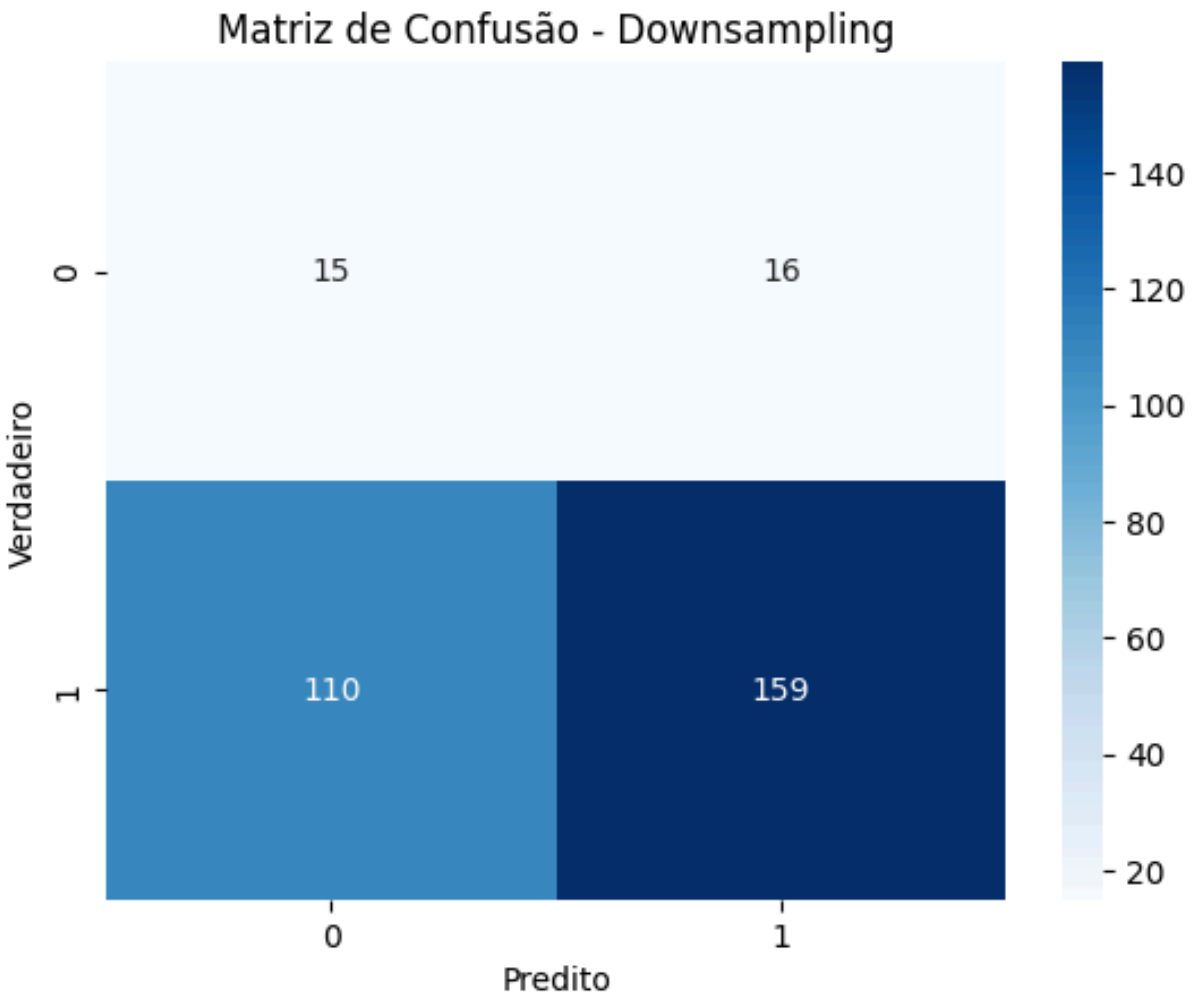

Figura 10.8: Matriz de confusão downsampling

E o relatório de classificação pode ser visto a seguir:

```
Relatório de Classificação:
              precision    recall  f1-score   support

    Negativo       0.12      0.48      0.19        31
    Positivo       0.91      0.59      0.72       269

    accuracy                           0.58       300
   macro avg       0.51      0.54      0.45       300
weighted avg       0.83      0.58      0.66       300
```

Vamos analisar o relatório de classificação que fizemos com os dados submetidos ao tratamento de *downsampling*:

Precisão (Precision)

- Negativo: 0.12 (a precisão, mesmo que diferente de 0, é baixa, indicando que o modelo está fazendo muitas previsões falsas positivas para esta classe).
- Positivo: 0.91 (a precisão para a classe `Positivo` continua alta, mostrando que o modelo é preciso ao prever a classe majoritária).

Revogação (Recall)

- Negativo: 0.48 (o recall para a classe `Negativo` melhorou em comparação com os dados desbalanceados e o *upsampling*, mas ainda não é ideal, sugerindo que o modelo está capturando uma quantidade moderada de casos da classe `Negativo`).
- Positivo: 0.59 (para a classe `Positivo` o recall é menor do que nos outros cenários, indicando que o modelo está perdendo alguns casos da classe majoritária).

F1-Score:

- Negativo: 0.19 (a F1-Score para a classe `Negativo` é baixa, refletindo um desempenho fraco na previsão dessa classe).
- Positivo: 0.72 (a F1-Score para classe `Positivo` é razoável, mas mais baixa do que nos cenários anteriores).

Acurácia: 0.58 (a acurácia baixa de 0.58 após o *downsampling* reflete uma diminuição na taxa geral de acertos. Entretanto, o modelo começou a identificar alguns casos da classe `Negativo` com um recall de 0.48. Isso indica que, apesar de uma acurácia menor, o modelo agora consegue capturar uma quantidade moderada de casos da classe minoritária, mostrando um equilíbrio entre as classes, embora ainda haja espaço para melhorias no desempenho geral).

Conclusões

1. Viés com dados desbalanceados: O modelo mostra uma alta acurácia, mas esse valor é ilusório porque o modelo não está prevendo corretamente a classe minoritária (`Negativo`). Isso é um exemplo claro de viés para a classe majoritária.
2. Efeito do *upsampling*: o *upsampling* melhora a revogação e a F1-Score para a classe majoritária, mas não resolve o problema da previsão da classe minoritária. Embora, a acurácia global tenha diminuído ligeiramente, o modelo está mais equilibrado em termos de prever a classe minoritária, o que pode ser mais útil em aplicações onde a classe minoritária é crítica.
3. Efeito do *downsampling*: o *downsampling* prorporciona um modelo mais equilibrado em termos de previsão das duas classes, entretanto o custo é significativo na acurácia geral. Esse método pode ser útil para problemas onde é essencial identificar corretamente ambas as classes, mesmo que isso signifique perder a acurácia geral.

Com isso, concluímos este livro. No apêndice a seguir, abordaremos o uso de métricas de classificação para avaliar e selecionar o melhor modelo e as transformações mais eficazes para os nossos dados,

com o objetivo de maximizar a precisão e a aplicabilidade dos resultados.

Apêndice

Capítulo 11
Apêndice: Qual é o melhor modelo?

> Você pode encontrar o material do apêndice neste link: materiais da parte 6 (https://github.com/ThiagoBenevides/feature-engine-transformando-dados-brutos-em-recursos/tree/main/parte_seis)

Até agora, aprendemos várias técnicas de engenharia de recursos que nos auxiliam a melhorar a qualidade dos nossos dados para serem usados em modelos de *machine learning*. Entretanto, é importante saber como avaliar a qualidade desses modelos para entendermos quão boas são nossas transformações.

Neste apêndice, vamos focar nas métricas de avaliação para modelos de classificação e de regressão. Veremos as métricas mais comuns para modelos de classificação, como acurácia, precisão, *recall*, F1-Score e AUC-ROC, e para modelos de regressão, como RMSE, MAE e R^2.

Além disso, falaremos sobre como essas métricas podem ser afetadas pelas transformações que fizemos em nossos dados. Compreender as métricas de avaliação é crucial para avaliar a qualidade dos modelos de *machine learning* e para tomar decisões informadas sobre quais modelos usar e quais transformações aplicar.

11.1 Métricas para avaliar modelos de classificação

Vamos começar nosso estudo com as métricas de classificação. Para isso, vamos utilizar os dados do Titanic com as transformações que já fizemos.

Compreendendo a acurácia

A acurácia é uma das métricas mais comuns e simples para avaliar a qualidade de um modelo de classificação. A acurácia mede a proporção de previsões corretas em relação ao número total de previsões realizadas pelo modelo.

Para calcularmos o valor da acurácia, basta somar o número de previsões corretas e dividir pelo número total de previsões, da seguinte forma:

```
Acurácia = (VP+VN)/(VP+FP+FN+VN)
```

Onde:

- `VP` : Verdadeiros positivos (são positivos e foram classificados como positivos);
- `VN` : Verdadeiros negativos (são negativos e foram classificados como negativos);
- `FP` : Falsos positivos (são negativos, mas foram classificados como positivos);
- `FN` : Falsos negativos (são positivos, mas foram classificados como negativos).

Apesar de a acurácia ser uma métrica útil e fácil de entender e interpretar, ela pode não ser o suficiente para avaliar a qualidade de um modelo, especialmente em casos de dados desbalanceados, em que a proporção entre as classes é muito desigual. Nesses casos, o modelo pode ter uma alta acurácia simplesmente prevendo a classe majoritária em todas as instâncias.

Então, para este exemplo utilizaremos o dataframe chamado `titanic_tratado.csv` , com o qual vamos construir uma árvore de

decisão para fazermos a previsão de quem sobreviveu e quem não sobreviveu.

Vamos começar importando as bibliotecas necessárias e o nosso arquivo CSV com o código a seguir:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from yellowbrick.classifier import ConfusionMatrix

titanic = pd.read_csv("titanic_tratado.csv")
titanic["Age"].fillna(titanic["Age"].mean(), inplace=True)
titanic.head()
```

O resultado pode ser visto na tabela a seguir:

Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Em
0	3	1	22	1	0	7.2	2
1	1	0	38	1	0	71.2	0
1	3	0	26	0	0	7.9	2
1	1	0	35	1	0	53.1	2
0	3	1	35	0	0	8.0	2

Observação: esse dataframe do Titanic já foi tratado previamente para estar em conformidade com o algoritmo de decisão.

Agora, vamos compreender um pouco melhor o código que fizemos. No primeiro trecho, fizemos as importações necessárias, incluindo a biblioteca *yellowbrick*, que nos ajuda a visualizar de forma mais simples as métricas dos algoritmos de *machine learning*. Em seguida, substituímos os valores ausentes da coluna `Age` pela média da coluna.

CONFUSIONMATRIX OU MATRIZ DE CONFUSÃO

No código anterior, você pode ter notado uma palavra que ainda não usamos, que é a `ConfusionMatrix`, em uma tradução livre, matriz de confusão. A matriz de confusão apresenta de forma tabular a quantidade de previsões corretas e incorretas geradas pelo algoritmo de *machine learning*. Ela possui duas dimensões principais, que são as classes dos valores verdadeiros e a classe dos valores previstos pelo modelo, sendo composta por quatro compartimentos: Verdadeiro Positivo, Verdadeiro Negativo, Falso Positivo e Falso Negativo.

```
X_treino,X_teste,y_treino,y_teste =
train_test_split(titanic.drop('Survived',axis =
1),titanic['Survived'],test_size = 0.3,random_state = 42)
```

Nas linhas do código anterior, realizamos a divisão do nosso conjunto de dados em conjuntos de treino e teste. Isso é feito separando a variável alvo `Survived` das demais variáveis preditoras. O código não mostra explicitamente a codificação da variável `Sex`, mas, considerando que a tabela de dados exibida já mostra `Sex` como valores numéricos, podemos inferir que essa etapa de codificação (transformar valores categóricos em numéricos) já foi realizada anteriormente no processo de tratamento dos dados. Agora, após a divisão em conjuntos de treino e teste, estamos prontos para criar e avaliar o nosso modelo de árvore de decisão.

```
from sklearn.tree import DecisionTreeClassifier

modelo = DecisionTreeClassifier(random_state = 42)

modelo.fit(X_treino,y_treino)
```

Com isso, conseguimos treinar o nosso modelo. Vamos fazer as previsões e calcular a acurácia utilizando a biblioteca *yellowbrick*:

```
previsores = modelo.predict(X_teste)

cm = ConfusionMatrix(modelo, classes =["Não sobreviveu", "Sobreviveu"])
cm.score(X_teste,y_teste)
cm.show()
```

A saída do código que acabamos de executar exibe a matriz de confusão do modelo. A matriz de confusão é uma tabela que mostra as contagens de previsões corretas e incorretas para cada classe. Nessa matriz, os verdadeiros positivos e verdadeiros negativos, que são as previsões corretas, aparecem na diagonal principal (do canto superior esquerdo para o inferior direito). Os falsos positivos e falsos negativos, que são as previsões incorretas, estão fora dessa diagonal. A acurácia do modelo é calculada somando os verdadeiros positivos e verdadeiros negativos e dividindo pelo total de previsões.

Com o código que acabamos de fazer, teremos como resposta a seguinte matriz:

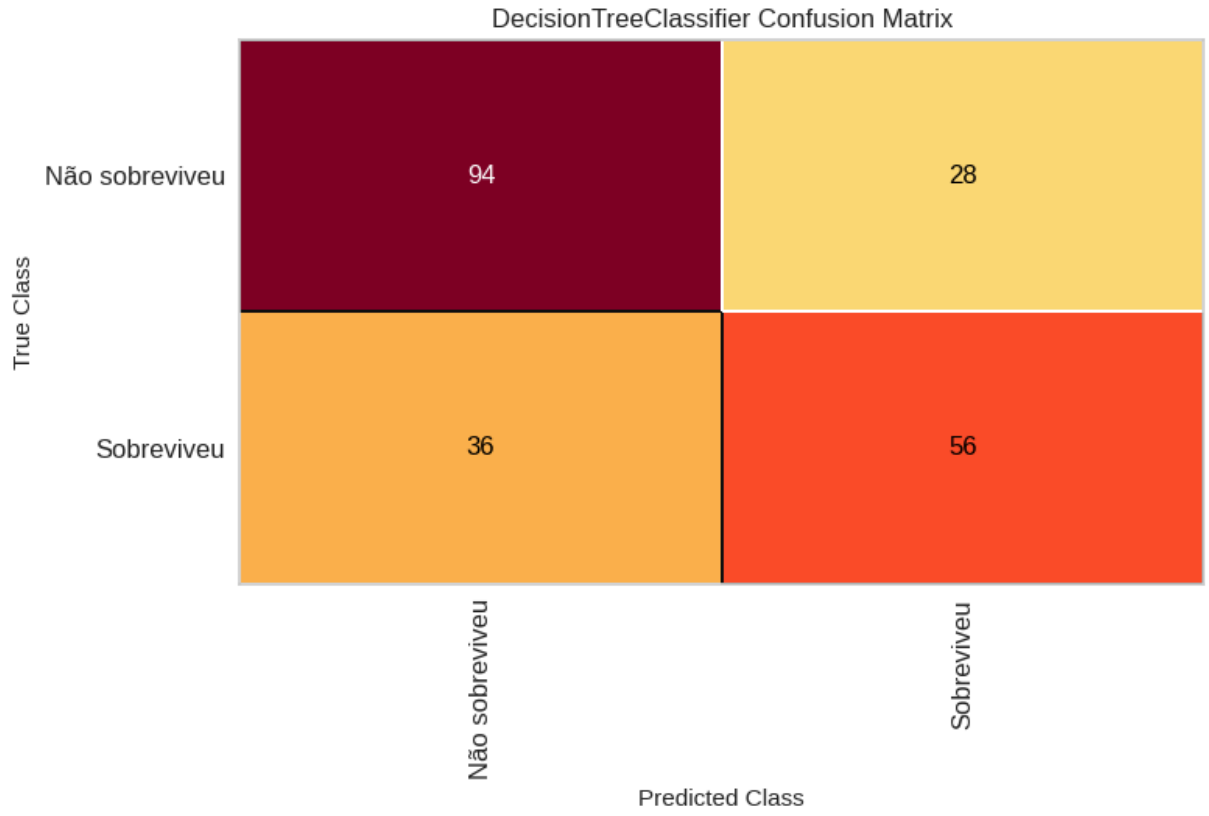

Figura 11.1: Matriz de confusão.

Como você pode observar, os valores na diagonal principal (94 e 56) representam as previsões corretas, enquanto os valores fora da diagonal (28 e 36) representam as previsões incorretas.

A matriz de confusão que criamos possui duas dimensões. No eixo y, temos o eixo `True Class`, que são os valores reais, ou seja, os valores que foram obtidos do nosso dataframe; já no eixo x, temos o eixo `Predicted Class`, que são os valores reais, ou seja, os valores previstos pelo algoritmo de *machine learning*. Então, temos os seguintes valores:

- Verdadeiro positivo: Sobreviveu e o algoritmo previu que sobreviveu (36 pessoas);
- Verdadeiro negativo: Não sobreviveu e o algoritmo previu que não sobreviveu (94 pessoas);

- Falso positivo: Não sobreviveu e o algoritmo previu que sobreviveu (28 pessoas);
- Falso negativo: Sobreviveu e o algoritmo previu que não sobreviveu (56 pessoas).

Com base no que acabamos de ver, conseguimos concluir alguns fatos a respeito de nossa árvore de decisão:

1. O algoritmo teve uma taxa considerável ao prever de forma assertiva os casos em que os passageiros não conseguiram sobreviver ao desastre (verdadeiros negativos);
2. O modelo apresentou dificuldades em prever corretamente os casos em que os passageiros conseguiram sobreviver (verdadeiros positivos). Embora tenha acertado em alguns casos, houve uma quantidade significativa de casos em que o algoritmo classificou de forma errada os sobreviventes como não sobreviventes (falso negativo);
3. O algoritmo também cometeu erros ao prever que passageiros não sobreviveram quando, na verdade, sobreviveram (falsos positivos), embora essa quantia tenha sido menor em comparação aos verdadeiros negativos.

Entretanto, é importante lembrar que a acurácia pode não ser um bom indicador de desempenho para todos os tipos de problemas de classificação. Isso ocorre porque a acurácia não leva em consideração os dados desbalanceados das classes e o custo de prever falsos positivos ou falsos negativos nos mais diversos cenários. Portanto, é fundamental utilizarmos outras métricas de avaliação para completarmos a análise de nosso modelo.

Compreendendo a precisão

A precisão é uma métrica utilizada em problemas de classificação e representa a proporção de observações classificadas como verdadeiras positivas em relação ao total de observações

classificadas como positivas. Para calcular a precisão, temos a seguinte fórmula:

```
Precisão = VP/(VP+FP)
```

Onde:

- `VP` : Verdadeiros positivos (são positivos e foram classificados como positivos);
- `FP` : Falsos positivos (são negativos e foram classificados como positivos).

A precisão é uma métrica importante, pois nos permite avaliar a capacidade do modelo de classificar corretamente as observações positivas, o que é especialmente relevante em problemas em que os falsos positivos podem ter consequências graves, por exemplo, detecção de fraudes e doenças.

Entretanto, caso existam muitos falsos negativos, a precisão pode ser baixa mesmo quando o modelo apresenta uma boa capacidade de classificação. Nesses casos, pode ser interessante utilizar outras métricas, que veremos mais adiante neste capítulo.

Para calcularmos a precisão, vamos utilizar o código a seguir:

```
from sklearn.metrics import precision_score

y_previsao = modelo.predict(X_teste)
precisao = precision_score(y_teste, y_previsao)
```

No código anterior, começamos importando da biblioteca `sklearn.metrics` o método `precision_score` , que vai nos auxiliar a calcular a precisão. Em seguida, fizemos a previsão de quem sobreviveu e de quem não sobreviveu utilizando o método `predict` passando como parâmetro o dataframe `X_teste` . Por fim, utilizamos o `precision_score` passando como parâmetros os valores reais em `y_teste` e os valores previstos pelo modelo que estão alocados em `y_previsao` .

Como resultado, recebemos o valor de 0.66, ou seja, nosso algoritmo teve uma precisão de 66%. Porém, vale ressaltar que, no contexto da precisão, estamos avaliando o desempenho do modelo em conseguir prever corretamente as amostras positivas, ou seja, a proporção de verdadeiros positivos em relação ao total de amostras classificadas como positivas (verdadeiros positivos mais falsos positivos).

Compreendendo o recall

O *recall*, também conhecido como sensibilidade, é uma métrica utilizada em problemas de classificação para avaliar a capacidade do modelo de identificar corretamente todas as instâncias de uma determinada classe. Em outras palavras, o recall mede a proporção de verdadeiros positivos em relação ao número total de instâncias que realmente pertencem àquela classe. A fórmula para calcular o recall é:

```
Recall = VP/(VP+FN)
```

Onde:

- `VP` : Verdadeiros positivos (valores positivos que são classificados como positivos);
- `FN` : Falsos negativos (valores positivos que são classificados como negativos).

O recall é especialmente importante em problemas em que a classe positiva é rara e/ou possui um impacto significativo na tomada de decisão. Por exemplo, em um modelo para detectar fraudes em transações financeiras, é fundamental que o recall seja alto para garantir que todas as transações fraudulentas sejam identificadas. Aqui vamos utilizar o *recall* nos nossos dados do Titanic para conseguirmos comparar com as outras duas métricas.

Vamos começar lendo o arquivo `titanic_tratado.csv` :

```
df = pd.read_csv("titanic_tratado.csv")
```

Em seguida, vamos selecionar as colunas relevantes e criar as variáveis de entrada `X` e de saída `y`:

```
X = df[["Sex", "Pclass"]]
y = df["Survived"]
```

Nos capítulos anteriores, realizamos os tratamentos necessários para preparar os dados para o modelo de *machine learning*. As variáveis categóricas, como `Sex`, já foram convertidas em valores numéricos, e as demais variáveis, incluindo `Pclass`, encontram-se no formato adequado para aplicação do modelo.

```
from sklearn.model_selection import train_test_split

# Codificar apenas 'Sex' se 'Pclass' já estiver em formato numérico
X = pd.get_dummies(X, columns=["Sex"])

# Se 'Pclass' precisar de codificação, descomente a linha abaixo
# X = pd.get_dummies(X, columns=["Sex", "Pclass",])

X_treino, X_teste, y_treino, y_teste = train_test_split(X, y,
test_size=0.3, random_state=42)
```

Assim, garantimos que todas as variáveis categóricas estejam corretamente tratadas e os dados preparados para a construção e avaliação do modelo. Em seguida, vamos criar o modelo e fazer a previsão usando um modelo de regressão logística para calcularmos o `recall_score`.

```
from sklearn.linear_model import LogisticRegression

modelo = LogisticRegression()
modelo.fit(X_treino,y_treino)

previsoes = modelo.predict(X_teste)
```

O que é o `LogisticRegression` que acabamos de instanciar? `LogisticRegression`, ou, em uma tradução livre, *regressão logística*, é um algoritmo de aprendizado supervisionado usado para resolver

problemas de classificação binária, ou seja, quando temos que classificar um exemplo em duas categorias distintas.

O objetivo da regressão logística é encontrar a relação entre as variáveis independentes (características) e a variável dependente (classe), estimando a probabilidade de um exemplo pertencer a uma determinada classe. A regressão logística usa a função logística (também chamada de função sigmoide) para modelar a relação entre as variáveis independentes e a probabilidade de pertencer a uma classe específica. A função logística é uma curva em forma de "S" que transforma qualquer valor real em um valor entre 0 e 1. Isso permite interpretar a saída do modelo como a probabilidade de pertencer a uma determinada classe. Se a probabilidade for maior que um determinado limite (geralmente 0.5), o exemplo é classificado como pertencente à classe positiva; caso contrário, é classificado como pertencente à classe negativa.

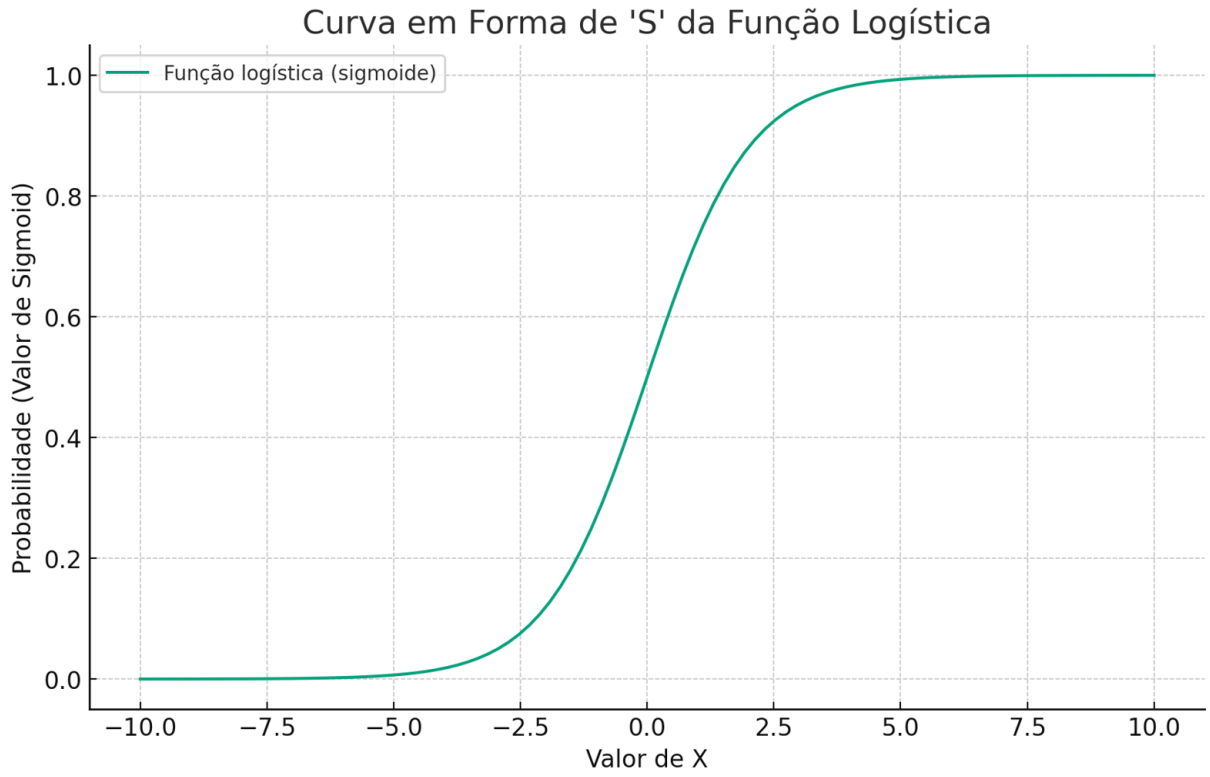

Figura 11.2: Ilustração da curva da regressão logística

Agora que construímos o modelo e entendemos o que a regressão logística faz, vamos calcular o recall do modelo com o seguinte código:

```
from sklearn.metrics import recall_score

recall = recall_score(y_teste,previsoes)
print("O valor do recall foi de: {:.2f}".format(recall))
```

O resultado da execução desse código é:

```
O valor do recall foi de: 0.67
```

Ou seja, o nosso modelo é capaz de identificar corretamente 67% dos casos positivos em relação ao total de casos positivos existentes.

Em outras palavras, o *recall* mede a taxa de verdadeiros positivos em relação ao número total de positivos reais. Em nosso contexto, isso significa que o modelo acerta cerca de 67% das vezes ao prever os exemplos que pertencem à classe positiva.

Compreendendo a F1-Score

A F1-Score é uma métrica calculada a partir de outras duas métricas, o recall e a precisão, em um único valor. Ela é útil para avaliar o desempenho de um modelo de classificação. O valor da F1-Score é calculado a partir da média harmônica descrita a seguir, que pode variar entre 0 a 1.

```
f1_core = (2*recall *precision)/(recall+precision)
```

Um valor próximo de 1 indica que o modelo teve um melhor desempenho, e o contrário, próximo de 0, indica que o modelo teve um desempenho ruim.

As principais vantagens da F1-Score são que ele leva em consideração tanto a precisão quanto o recall, o que o torna uma métrica mais equilibrada em relação à acurácia. Além disso, ele é útil

em situações em que as classes estão desbalanceadas, ou seja, quando uma classe é menor que outra.

Entretanto, tenha em mente que a F1-Score não leva em consideração os verdadeiros negativos, o que pode ser importante em alguns casos. Além disso, a F1-Score é sensível a desequilíbrios nas classes e pode não ser adequado a todos os casos.

Vamos calcular a F1-Score do nosso modelo de árvore de decisão do Titanic:

```
from sklearn.metrics import f1_score

f1 = f1_score(y_teste,previsoes)

print(f"O valor do F1-Score é de: {f1}")
```

Por fim, vamos visualizar essa métrica utilizando a biblioteca `yellowbrick`:

```
from yellowbrick.classifier import ClassificationReport

vis = ClassificationReport(modelo,support=True)
vis.fit(X_treino,y_treino)
vis.score(X_teste,y_teste)
vis.show()
```

Executando o código anterior, teremos a seguinte imagem como resultado:

Figura 11.3: Relatório com as métricas de desempenho do nosso modelo.

Na tabela da imagem anterior, temos a seguinte interpretação:

- A primeira linha representa a classe positiva (1);
- A segunda linha representa a classe negativa (0).

Os valores apresentados são os seguintes:

- `precision` : O valor de 0.795 indica a proporção de verdadeiros positivos em relação à soma dos verdadeiros positivos e falsos positivos;
- `recall` : O valor de 0.674 indica a proporção de verdadeiros positivos em relação à soma dos verdadeiros positivos e falsos negativos;
- `f1` : O valor de 0.729 é a média harmônica entre a precisão e o recall.

Além disso, temos as informações de suporte, que representam o número de ocorrências de cada classe em nosso conjunto de teste.

No caso da *F1-Score*, temos um valor de 72.9% que indica para nós que o modelo é capaz de classificar corretamente cerca de 72.9% (aproximadamente 73%) dos exemplos positivos, levando em consideração tanto a capacidade de identificar corretamente os exemplos positivos (recall) quanto a proporção de exemplos classificados corretamente entre os positivos previstos (precisão).

Compreendendo a AUC-ROC

A **ROC** (*Receiver Operanting Characteristic*) é a curva que avalia o desempenho de modelos de classificação binária, mostrando a relaçao entre a taxa de verdadeiros positivos (sensibilidade) e a taxa de falsos positivos (1-especificidade) para diferentes limiares de decisão.

Já a **AUC**(*Area Under the Curve*) é o valor numérico que representa a área sob a curva **ROC*. Ela varia de 0 a 1, onde valores mais próximos de 1 indicam que o modelo tem uma alta capacidade de distinguir entre as classes, enquanto valores próximos de 0.5 sugerem um modelo que não tem poder preditivo, equivalente a uma classificação aleatória.

Uma das vantagens da **AUC-ROC** é sua independência em relação ao limiar de decisão, o que a torna útil em cenários onde precisão e *recall* podem ser insuficientes, especialmente em conjuntos de dados desbalanceados. Entretanto, ela não considera a distribuição das classes, o que pode ser uma limitação em casos de desequilíbrios extremos.

Essa métrica é amplamente utilizada, pois permite avaliar o modelo em diferentes cenários e selecionar o limiar de decisão mais adequado às necessidades do problema. Por exemplo, em contextos críticos, como saúde, maximizar verdadeiros positivos e minimizar falsos negativos é essencial para evitar diagnósticos incorretos.

Entretanto, a AUC-ROC não leva em consideração a distribuição das classes, o que pode ser um problema em conjuntos de dados muito desbalanceados. Além disso, ela não fornece uma interpretação direta das taxas de erro do modelo.

Vamos calcular a AUC-ROC da nossa regressão logística:

```
from sklearn.metrics import roc_auc_score

auc_roc = roc_auc_score(y_teste,previsoes)

print("O resultado do AUC-ROC é de:" auc_roc)
```

Ao executarmos esse código, temos o seguinte resultado:

```
O resultado do AUC-ROC é de: 0.771382751247327
```

Nesse caso, conseguimos um valor de aproximadamente 77%, indicando que o nosso modelo de regressão logística apresentou um desempenho razoável na classificação dos exemplos nos dados de teste.

Por fim, vamos construir o gráfico com a curva AUC-ROC com o `yellowbrick`, com o seguinte código:

```
from yellowbrick.classifier import ROCAUC

visualizador = ROCAUC(modelo)

visualizador.fit(X_treino, y_treino)
visualizador.score(X_teste, y_teste)

visualizador.show()
```

No código que acabamos de executar, começamos importando o método `ROCAUC` da biblioteca `yellowbrick.classifier`. Em seguida, instanciamos o `ROCAUC` passando como parâmetro nosso modelo de regressão logística para ser avaliado e o salvamos na variável `visualizador`. Depois, ajustamos o `ROCAUC` aos dados de treino com o método `fit` e calculamos o *score* passando os dados de teste com o

método `.score`. Para finalizar, exibimos o gráfico com o comando `show`.

O resultado pode ser visto na imagem a seguir:

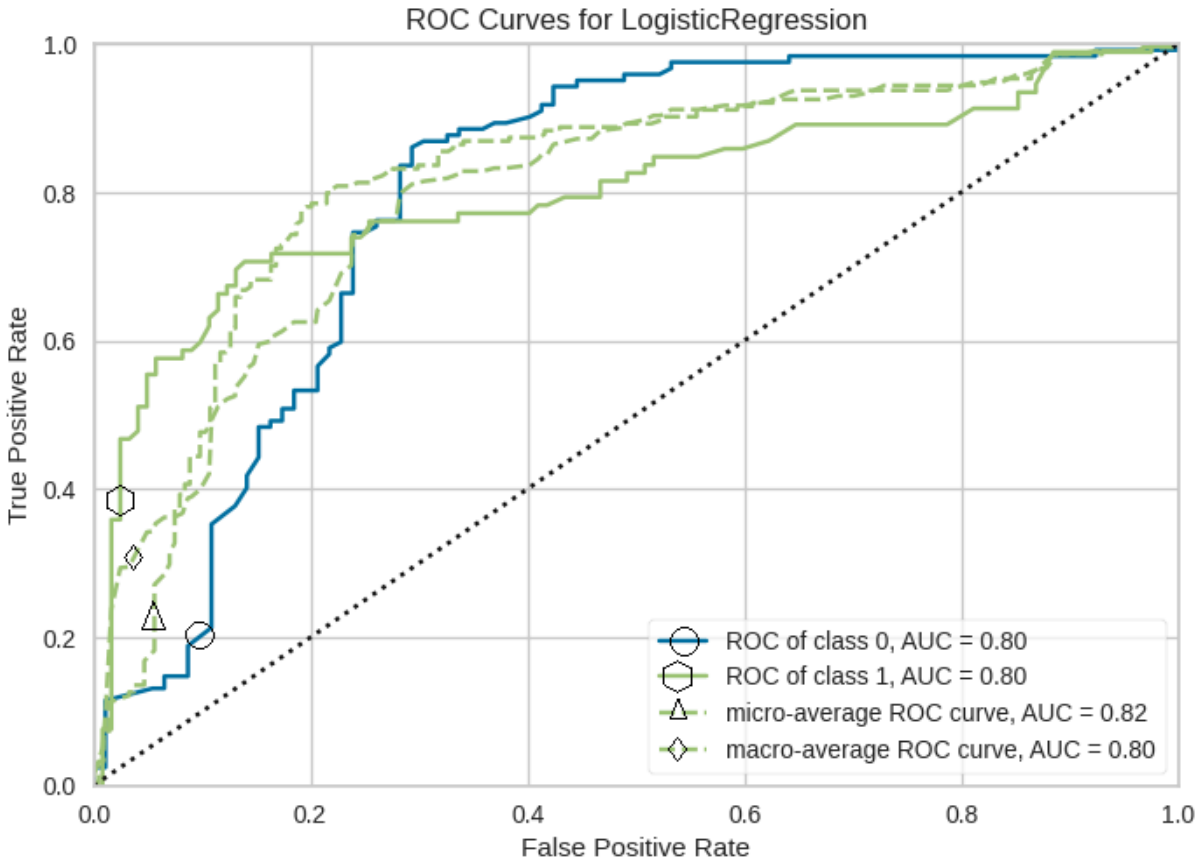

Figura 11.4: Resultado da curva AUC-ROC

Vamos compreender o que cada uma dessas curvas significa:

- `ROC of class 0, AUC = 0.80` : essa é a curva para a classe 0 do seu problema de classificação. O valor AUC igual a 0.80 indica uma boa capacidade do modelo em distinguir corretamente os exemplos de classe 0 dos exemplos de classe 1;
- `ROC of class 1, AUC = 0.80` : essa é a curva ROC para a classe 1 do seu problema de classificação. O valor AUC é novamente 0.80, indicando uma capacidade de distinguir corretamente os exemplos de classe 1 dos de classe 0;

- `Micro-average ROC curve, AUC = 0.82` : essa é a curva ROC calculada utilizando uma abordagem do `micro-average` . Ela considera todos os exemplos da classificação como um único conjunto, calculando a média das taxas de verdadeiros positivos e falsos positivos;
- `Macro-average ROC curve, AUC = 0.80` : essa é a curva calculada utilizando uma abordagem de `macro-average` . Ela calcula a média das curvas ROC para cada classe separadamente.

MICRO-AVERAGE E MACRO-AVERAGE

- Micro-average: todas as instâncias individuais são consideradas igualmente importantes. A métrica é calculada agregando as contagens e verdadeiros positivos, falsos positivos e falsos negativos de todas as classes e, então, é calculada a métrica desejada com base nesses totais agregados. Essa abordagem é especialmente útil para quando há um desequilíbrio significativo entre as classes do nosso conjunto de dados.
- Macro-average: a métrica é calculada separadamente para cada uma das nossas classes e então é calculada a média dessas métricas de forma individual para obtermos uma métrica única. Essa abordagem é útil para quando desejamos ter uma ideia do desempenho do modelo em cada uma das classes de forma individual e não dar mais peso às classes maiores ou menores.

Em relação ao modelo criado, nossa curva AUC-ROC indica que nosso modelo possui um bom desempenho de classificação, com AUCs relativamente elevadas para nossas classes e para a média global. Isso sugere que o modelo é capaz de fazer a distinção de forma correta entre as nossas classes e possui um bom desempenho na execução da tarefa de classificação.

Avaliando as diferentes métricas de classificação que obtivemos

Ao analisarmos as diferentes métricas obtidas com a árvore de decisão com os dados do Titanic, podemos observar que a acurácia foi de 66%, o que significa que o modelo classificou corretamente 66% dos dados de teste. Entretanto, a acurácia não leva em consideração o desbalanceamento das classes, o que pode ser problemático em um conjunto de dados como este, em que a maioria das pessoas não sobreviveu. Isso pode levar a uma interpretação enganosa dos resultados.

Ao analisarmos as outras métricas, como precisão, recall e F1-Score, podemos ver que elas levam em consideração a proporção de verdadeiros positivos, falsos positivos e falsos negativos na classificação, o que nos dá uma visão mais completa do desempenho do modelo. O recall de 67% indica que o modelo identificou corretamente 67% dos indivíduos que realmente sobreviveram. O F1-Score, que é a média harmônica da precisão e do recall, atingiu um valor de 72%, indicando um desempenho equilibrado em termos de identificar corretamente tanto os sobreviventes quanto os não sobreviventes.

Como analistas de dados, devemos levar em consideração todas essas métricas ao avaliar o desempenho do modelo para decidir qual é a melhor métrica para o nosso uso específico. No caso do Titanic, pode ser mais importante maximizar o recall, para garantirmos a identificação correta de todos os sobreviventes, mesmo que isso signifique uma taxa maior de falsos positivos. Entretanto, isso pode depender do contexto específico do problema e das implicações de falsos positivos e falsos negativos.

Agora que já entendemos como calcular e interpretar as principais métricas de avaliação de modelos de classificação, é importante destacar os pontos-chaves aos quais devemos estar atentos ao analisarmos o desempenho de um modelo. A escolha da métrica certa depende do problema em questão, do impacto dos erros e do

equilíbrio entre as classes no conjunto de dados. Vamos revisar os aspectos mais importantes de cada métrica e como elas se complementam na avaliação geral do modelo:

- **Acurácia**: neste caso, o modelo atingiu 66% de acurácia, indicando que classificou corretamente 66% dos dados de teste. Contudo, a acurácia pode ser enganosa em dados desbalanceados, como no caso do Titanic, em que a maioria das pessoas não sobreviveu
- **Precisão**: reflete a proporção de previsões corretas entre todas asprevisões positivas. É relevante quando queremos minimizar falsos positivos, mas não é o foco principal neste caso.
- **Recall**: com 67%, indica que o modelo identificou corretamente 67% dos sobreviventes. O *recall* é importante quando olhamos para cenários onde é mais importante evitar falsos negativos (não indetificar sobreviventes reais neste caso)
- **F1-Score**: atingiu 72%, representando um equilíbrio entre precisão e *recall*. Esta métrica é útil quando ambas as metricas precisão e *recall* são igualmente importantes.

Em nosso caso em particular, pode ser mais relevante priorizar o *recall* para garantirmos que o máximo de sobreviventes sejam indetificados, mesmo ao custo de um número maior de falsos positivos.

11.2 Métricas para avaliar modelos de regressão

Nesta seção, veremos mais métricas para avaliarmos algoritmos de *machine learning* após nossas transformações. Aqui focaremos nos algoritmos de regressão, ou seja, algoritmos que tentam de certa forma prever resultados numéricos. Veremos algumas das métricas

mais importantes para avaliar modelos de regressão, incluindo o RMSE (erro médio quadrático), o MAE (erro absoluto médio) e o R^2 (coeficiente de determinação). Discutiremos como cada uma dessas métricas funciona, como interpretá-las e em quais cenários podem ser úteis. Também abordaremos a importância de compreender o contexto do problema de regressão em que se está trabalhando, para que você consiga escolher a métrica mais apropriada para avaliar seu modelo.

Compreendendo o RMSE

O **RMSE** é uma das métricas mais utilizadas para a avaliação do algoritmo de regressão. Essa métrica é especialmente útil quando queremos avaliar o quanto nosso modelo conseguiu se ajustar aos dados observados, e é particularmente útil em casos em que a distribuição dos dados é simétrica. O RMSE é definido como a **raiz quadrada da média dos erros quadráticos** entre os valores previstos pelo modelo e os valores reais dos nossos dados.

Raiz quadrada da média dos erros quadráticos

Pode parecer confuso, mas não é. Veja só: primeiro, calculamos o erro entre os valores previstos pelo modelo e os valores reais dos dados. Em seguida, elevamos esses erros ao quadrado para eliminar os sinais negativos e enfatizar os erros maiores. Em seguida, calculamos a média desses erros quadráticos, somando todos eles e dividindo pelo número total de observações. Para finalizar, aplicamos a operação de raiz quadrada para obtermos uma medida do erro médio, que é uma forma de estimar quão próximos os valores previstos estão dos valores reais.

Para entender melhor, vamos a um exemplo. Vamos supor que os valores previstos são `previsao = [4,7]` e que os valores reais são `reais = [3,6]`. Primeiro vamos calcular o erro quadrático para o primeiro valor e depois para o segundo valor, da seguinte forma:

- Primeiro valor: $(4-3)^2 = 1$

- Segundo valor: $(7-6)^2 = 1$

Em seguida, calculamos a média desses erros quadráticos:

```
Média dos erros quadráticos: (1+1)/2=1
```

Por fim, aplicamos a raiz quadrada a essa média para obtermos o RMSE:

```
rmse = np.sqrt(1)=1
```

Portanto, nesse exemplo hipotético, o valor do RMSE é igual a 1. Isso significa que, em média, as previsões do modelo diferem em 1 unidade dos valores reais.

Uma das principais vantagens em utilizarmos o RMSE é que ele nos fornece uma métrica quantitativa para avaliar a qualidade do ajuste do modelo, permitindo comparar os diferentes modelos para vermos qual é o mais preciso e confiável. Além disso, o RMSE é útil para a localização de possíveis valores discrepantes, uma vez que são elevados ao quadrado antes de serem somados. Isso significa que grandes erros são penalizados de forma mais acentuada.

Entretanto, devemos considerar que o RMSE é muito sensível a valores discrepantes. Logo, a utilização do RMSE deve ser repensada caso tenhamos muitos valores discrepantes em nossos dados.

Para nosso exemplo prático, utilizaremos o dataframe `mercado.csv`, que vimos na parte 2. Nós utilizaremos uma regressão linear para tentarmos prever o quanto uma pessoa vai gastar dada a sua renda. Para isso, utilizaremos o seguinte código:

```
import pandas as pd
from sklearn.linear_model import LinearRegression

mercado = pd.read_csv("mercado.csv")

X = mercado[['renda']]
y = mercado['gasto']
```

```
modelo = LinearRegression()
modelo.fit(X_treino, y_treino)
```

Na primeira linha, importamos a biblioteca `pandas` e o método `LinearRegression` da biblioteca `sklearn.linear_model`. Em seguida, carregamos nosso arquivo `mercado.csv` e fizemos a divisão em dados entre a variável que vamos utilizar para fazer a previsão, que em nosso caso é a variável `renda`, e a variável que tentaremos prever, que é a variável `gasto`. Para finalizar, instanciamos nosso modelo e ajustamos aos dados.

Vamos calcular o RMSE do nosso modelo com o código a seguir:

```
predicoes = modelo.predict(X_teste)
rmse = mean_squared_error(y_teste, predicoes, squared=False)

print(f"RMSE: {rmse}")
```

O que nós fizemos no código anterior foi fazer as previsões com o método `.predict`, passando como parâmetro o dataframe `X_teste`. Em seguida, fizemos o cálculo do RMSE com o método `mean_squared_error` passando como parâmetros o `y_teste` e as previsões, e finalizamos imprimindo na tela o valor do RMSE encontrado, que foi de 601.9787240279612, ou seja, aproximadamente R$ 601.98.

CUIDADO

Dentro do método `mean_squared_error` passamos o parâmetro `squared` como sendo `False`. Isso faz com que a função `mean_squared_error` retorne o RMSE. Caso você passe o `squared` como sendo `True`, não receberemos o RMSE e, sim, o MAE, que é o `mean_absolute error`.

Compreendendo o MAE

O **MAE**, ou *Mean Absolute Error*, é uma métrica de avaliação de modelos de regressão que calcula a média do valor absoluto das diferenças entre as previsões e os valores reais. Em outras palavras, ele mede a magnitude média dos erros.

A principal vantagem na utilização dessa métrica é a de que sua interpretação é intuitiva, pois nos fornece uma medida direta de quão bem o modelo está se ajustando aos dados. Além disso, o MAE é menos sensível a valores extremos do que o RMSE, pois não envolve a elevação dos erros ao quadrado.

No entanto, o MAE apresenta uma desvantagem em relação ao RMSE, que é o fato de não penalizar tanto os erros maiores. Isso pode ser problemático em certos cenários. Além disso, ele não é diferenciável em zero, o que pode dificultar sua utilização em algoritmos de otimização.

> Ser diferenciável em zero significa que a função de erro (no caso, o MAE) tem uma derivada contínua em zero. Isso é importante para algoritmos de otimização que usam o cálculo da derivada para encontrar o mínimo da função de erro.
>
> O MAE não é diferenciável em zero, pois a sua derivada não é bem definida em zero. Em outras palavras, não é possível calcular a derivada do MAE no ponto onde o erro é zero, e isso pode dificultar a aplicação do MAE em certos algoritmos de otimização que requerem o cálculo da derivada da função de erro.

Vamos utilizar o MAE no nosso algoritmo de regressão linear criado no tópico passado, em que calculamos o RMSE:

```
from sklearn.metrics import mean_absolute_error
```

```
mae = mean_absolute_error(X_teste,predicoes)
```

Como estamos utilizando o modelo de regressão linear feito no RMSE, não foi necessário refazer o modelo. Sendo assim, a única coisa que fizemos aqui foi importar o método `mean_absolute_error` da biblioteca `sklearn.metrics` para fazer o cálculo do erro. Ao executarmos o código anterior, teremos como resultado o seguinte:

```
O valor do MAE é de: 601.3407536017578
```

Ou seja, o erro foi de aproximadamente R$ 601.34 nas nossas previsões.

Compreendendo o R²

O R^2 é uma métrica utilizada para avaliar o quão bem os dados se ajustam à linha de regressão. Ele mede a proporção da variância dos dados que é explicada pelo modelo. O R^2 varia no intervalo `[0,1]`, sendo que valores mais próximos de 1 indicam que o modelo é capaz de explicar uma grande parte da variância dos dados e valores próximos de 0 indicam que o modelo não explica bem a variação dos nossos dados.

A principal vantagem em utilizarmos essa métrica é que ela é simples de ser interpretada. Um valor alto indica que o modelo é capaz de explicar grande parte da variação dos dados. Além disso, ela pode ser usada para comparar diferentes modelos, pois quanto maior o valor, melhor é o modelo.

Por outro lado, ele pode ser enganoso em certas situações. Por exemplo, se o modelo estiver superajustado aos nossos dados, o R^2 pode ser alto, mas o modelo pode não ser útil para fazer previsões precisas. Além disso, ele não leva em consideração a distribuição dos erros, ou seja, ele não indica se os erros estão distribuídos uniformemente ou se há erros grandes em algumas previsões.

Vamos calcular o R^2 do nosso algoritmo de regressão linear para os dados das ações, o mesmo usado no exemplo anterior:

```
from sklearn.metrics import r2_score

r2 = r2_score(y_teste,predicoes)
print(f"O valor de r2:{r2}")
```

Da mesma forma como fizemos no MAE, utilizamos o modelo de regressão linear, então a única coisa que fizemos foi importar o método `r2_score` e, em seguida, calcular o R^2 das nossas previsões com os valores reais. Se executarmos o código anterior, teremos o seguinte resultado:

```
O valor de r2:-161.07480114707062
```

> Você pode estar achando estranho esse valor de R^2 encontrado do nosso algoritmo se antes falamos que ele estava dentro de um intervalo fechado `[0,1]`. Realmente, esse valor é estranho e significa que o nosso modelo de regressão linear não é capaz de explicar a variabilidade dos nossos dados de forma adequada.

Avaliando as diferentes métricas de regressão que obtivemos

As três métricas de regressão que vimos nesta seção foram o RMSE, o R^2 e o MAE. Cada uma possui suas próprias características, e devemos estar atentos às suas particularidades ao escolhermos uma delas para avaliar a precisão do nosso modelo de previsão.

Ao utilizarmos o RMSE, obtivemos um valor de R$ 601.97 em nossas previsões. O RMSE é uma métrica que mede a raiz quadrada da média dos erros ao quadrado entre as previsões e os valores reais. Isso significa que ele penaliza os erros maiores de forma mais intensa, o que pode ser útil em alguns cenários. Porém, ele pode ser influenciado por valores extremos, o que pode afetar a interpretação.

O MAE obteve o valor de R$ 601.34. É uma métrica que mede a média dos erros absolutos entre as previsões e os valores reais. Porém, diferente do RMSE, o MAE não faz a penalização dos erros maiores de forma mais intensa, o que pode ser útil em alguns cenários de previsão. Porém, ele pode não ser tão sensível a mudanças nos valores das previsões, o que pode ser uma desvantagem em algumas situações.

Já o R^2 obteve o valor de R$ -161.07, mostrando que nosso modelo de regressão linear não consegue explicar a complexidade dos nossos dados, tendo em vista que o R^2 é uma métrica que mede a proporção da variação dos valores reais que é explicada pelo modelo. Essa métrica varia entre 0 e 1, sendo que valores mais próximos de 1 indicam um bom desempenho do modelo. É importante notar que o R^2 não é uma medida de erro absoluto, mas sim uma medida de quão bem o modelo se ajusta aos dados.

Em termos de escolha das métricas, é importante consideramos o objetivo, ou seja, o propósito de criação do modelo. Por exemplo, no caso do modelo de regressão linear que desenvolvemos, nosso objetivo era prever o valor gasto pelo cliente com base em sua renda que mais tarde essa previsão poderia ser utilizada para o direcionamento de propagandas ou descontos para estimular suas compras.

Com base nas métricas que foram utilizadas, podemos concluir que nosso modelo de regressão linear não é o mais adequado para prever os valores de gasto com base na renda do cliente. O RMSE obteve um valor de R$ 601.97, enquanto o MAE apresentou um valor de R$ 601.34, ambos indicando erros consideráveis em nossas previsões. Além disso, o valor do R^2 foi de -161.07, confirmando nossa hipótese de que o modelo não é adequado. Essas três métricas em conjunto evidenciam a falta de ajuste e precisão do modelo em relação aos dados observados.

Sendo assim, é importante compreender a função de cada uma das métricas que vimos nesta seção:

- O RMSE (*Root Mean Squared Error*) é utilizado quando nosso objetivo é minimizar os erros maiores, ou seja, penalizar mais intensamente os erros de maior magnitude;
- O MAE (*Mean Absolute Error*) é utilizado quando nosso objetivo é minimizar os erros de forma geral, sem uma penalização diferenciada para erros maiores ou menores;
- O R^2 (*R-squared*) é usado para avaliar o desempenho geral do modelo, medindo a proporção da variação dos valores reais que é explicada pelo modelo. Valores mais próximos de 1 indicam um bom ajuste do modelo aos dados.

Com a escolha adequada da métrica, podemos avaliar de forma mais precisa o desempenho do modelo e alinhar os resultados às nossas metas e objetivos.

11.3 Conclusão

Chegamos ao fim deste livro, mas para você, leitor ou leitora, é apenas o começo de uma jornada emocionante no mundo da análise de dados. Espero sinceramente que tenha gostado de aprender sobre esses conceitos fundamentais e que este livro tenha proporcionado muitos insights e ideias para o seu futuro no campo da análise de dados.

Parabéns por ter chegado até aqui, e que seu caminho seja repleto de descobertas e oportunidades incríveis. Desejo-lhe sucesso em suas empreitadas e obrigado por me acompanhar até aqui!